Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire

Frédéric Guidec

N° d'ordre : 1390

THÈSE

Présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

Pour obtenir

Le Titre de Docteur de l'Université de Rennes 1
Mention INFORMATIQUE

par

Frédéric GUIDEC

Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire

Soutenue le 2 juin 1995 devant la Commission d'examen

MM. :	Jean-Pierre	BANÂTRE	Président
	Jean-Marc	GEIB	Rapporteurs
	Denis	TRYSTRAM	
Mme :	Françoise	ANDRÉ	Examinateurs
MM. :	Jean-Marc	JÉZÉQUEL	
	Bernard	PHILIPPE	

A mes parents.

Remerciements

Je remercie M. Jean-Pierre BANÂTRE, *professeur à l'université de Rennes 1 et directeur de l'IRISA, qui me fait l'honneur de présider ce jury.*

Je remercie Mme Françoise ANDRÉ, *professeur à l'université de Rennes 1, d'avoir dirigé mon travail de thèse, et M. Jean-Marc* JÉZÉQUEL, *chargé de recherche au CNRS, pour m'avoir soutenu, orienté et conseillé tout au long de ce travail.*

Je remercie M. Denis TRYSTRAM, *professeur à l'Institut National Polytechnique de Grenoble, et M. Jean-Marc* GEIB, *professeur à l'université de Lille et directeur du Laboratoire d'Informatique Fondamentale de Lille, d'avoir bien voulu accepter la charge de rapporteurs.*

Je remercie M. Bernard PHILIPPE, *professeur à l'université de Rennes 1, d'avoir accepté de juger ce travail.*

Je remercie le Conseil Régional de Bretagne pour avoir financé mon travail conjointement avec l'INRIA, dans le cadre de la convention de financement $n^o 492C243003131806$.

Je remercie enfin tous les membres du projet PAMPA (Programmation des architectures parallèles réparties : fondements et méthodologie) pour leur aide amicale permanente.

Table des matières

Introduction

Les architectures parallèles à mémoire distribuée (APMD) sont des super-calculateurs dont les performances peuvent aujourd'hui atteindre plusieurs dizaines de Gflops[1]. Une APMD est constituée de plusieurs centaines, voire plusieurs milliers de nœuds, interconnectés par l'intermédiaire d'un réseau de communication. Chaque nœud comporte une unité de calcul, une unité de mémoire locale, et une unité de communication. Un nœud peut donc procéder à des calculs de manière autonome, et échanger des données avec les autres nœuds du réseau. Grâce à cette structure modulaire et extensible, une APMD peut être étendue de manière à fournir une puissance de calcul — théoriquement — illimitée.

Cette caractéristique ne pouvait manquer d'attirer l'attention des programmeurs scientifiques[2], toujours désireux d'accroître le champ d'investigation de leurs études grâce à une puissance de calcul accrue. Cependant, bien que le monde physique qu'ils modélisent soit foncièrement parallèle, les programmeurs scientifiques ont coutume de s'appuyer sur des techniques et des algorithmes séquentiels pour résoudre leurs problèmes. En fait, l'intérêt dont ils font preuve envers les architectures parallèles en général, et envers les APMD en particulier, résulte uniquement d'un désir d'améliorer les performances de leurs programmes d'application lorsque ceux-ci nécessitent une très grande puissance de calcul. Le non-déterminisme inhérent à l'exécution de programmes répartis sur APMD, par exemple, apparaît à leurs yeux comme un effet indésirable de l'exécution répartie plutôt que comme un atout potentiel [102].

À ce jour, la diffusion des APMD dans la communauté scientifique demeure pourtant très limitée, principalement en raison du manque de maturité des outils logiciels associés à ces machines. Les méthodes et les environnements de programmation adaptés aux machines mono-processeur traditionnelles s'avèrent inutilisables

1. 1 Gflops (giga-flops) : un milliard d'opérations en virgule flottante par seconde.

2. Nous désignons par ce terme les programmeurs qui, bien que n'étant pas informaticiens de formation, sont amenés à développer ou à utiliser des programmes d'application pour résoudre les problèmes rencontrés dans leur activité principale (qui peut relever du domaine de la physique, de la chimie, etc.).

avec les APMD, car ils ne permettent pas d'en maîtriser le parallélisme. L'environnement logiciel des APMD aujourd'hui disponibles est la plupart du temps constitué de bibliothèques de routines permettant de gérer la communication entre processus décrits dans des langages séquentiels tels que Fortran, C ou Lisp. Pour exploiter réellement une APMD, un programmeur doit alors posséder, non seulement une connaissance approfondie du domaine d'application considéré, mais aussi être capable d'exprimer un algorithme réparti en termes de processus communicants, connaître les spécificités de l'architecture utilisée et de son système d'exploitation. Le code obtenu avec une telle approche est difficile à développer, à comprendre, à mettre au point et à maintenir.

Les tâches de parallélisation des calculs, de répartition des données, et de gestion des processus et des communications n'ayant rien de spécialement attrayant, les programmeurs scientifiques sont en général assez réticents à l'idée de devoir porter manuellement leurs applications sur des APMD. C'est pourquoi de nombreuses recherches actuelles visent à simplifier le portage des applications scientifiques sur des APMD.

Plusieurs modèles de programmation parallèle ont été proposés à cette fin, qui s'efforcent tous de masquer à l'utilisateur la structure réelle de l'architecture parallèle utilisée, et les mouvements de données au sein de cette structure.

Dans le modèle SIMD (*Single Instruction Multiple Data*), on préserve le flot de contrôle unique du modèle séquentiel, mais chaque instruction peut être appliquée concurremment à des données situées sur des nœuds différents. Ce modèle permet de paralléliser aisément les algorithmes dans lesquels les calculs s'expriment en termes d'opérations vectorielles. Des machines SIMD telles que la Connection Machine ont d'ailleurs été construites pour traiter les problèmes de ce type.

Le domaine d'application des machines SIMD étant cependant trop restreint, la plupart des APMD développées récemment sont des machines de type MIMD. Chaque nœud a son propre flot de contrôle, et on peut donc potentiellement charger et exécuter un programme différent sur chacun des nœuds d'une machine MIMD.

D'importants travaux sont en cours, visant à la parallélisation totalement automatique de programmes séquentiels de type Fortran. Cependant, l'expérience montre que la parallélisation automatique pour APMD est une tâche ardue. Il faut tout d'abord identifier les dépendances entre les données manipulées dans un programme d'application, et il faut ensuite répartir les données de manière judicieuse sur les nœuds de l'APMD cible en tenant compte de ces dépendances, afin d'obtenir le maximum de concurrence à l'exécution tout en minimisant les communications. À ce jour, les techniques d'identification des dépendances sont relativement bien maîtrisées — du moins pour les problèmes réguliers [123, 117, 49, 50] —, mais la

distribution totalement automatique des données demeure un problème ouvert qui motive encore de très nombreuses recherches [43, 90, 100, 105].

Dans l'approche préconisée dans le projet Athapascan [103], le code est structuré comme un graphe d'appels de procédures. L'objectif de ce projet est de parvenir à traiter ce graphe automatiquement de façon à rendre efficace la répartition des calculs et les mouvements de données. La répartition de charge automatique doit être paramétrée par les caractéristiques de granularité de la machine cible. Cette paramétrisation peut être statique ou dynamique si l'environnement est évolutif (par exemple dans le cas d'une plate-forme constituée d'un réseau de stations de travail). Le noyau exécutif parallèle qui constitue le support d'exécution du projet Athapascan (niveau Athapascan-0) est constitué d'un ensemble de serveurs capables de répondre aux appels de procédures à distance de manière synchrone ou asynchrone. L'approche est donc SAMD (*Single Application Multiple Data*), chaque processeur pouvant exécuter soit le même code (lorsque les serveurs ne sont pas spécialisés), soit un code différent des autres processeurs.

Avec le modèle SPMD, on tente de prendre en compte le fait que la plupart des problèmes qu'on souhaite résoudre sur des APMD sont caractérisés par le volume important des données à traiter. Le modèle SPMD (*Single Program Multiple Data*) préserve la simplicité conceptuelle du modèle SISD, et bénéficie du parallélisme du modèle SIMD. Deux approches sont envisageables pour la mise en œuvre de programmes SPMD : on peut baser la parallélisation sur une distribution du contrôle (par découpage des boucles), ou sur une distribution des données.

La première approche nécessite que soit implanté sur l'APMD utilisée un système de mémoire virtuelle partagée. Le compilateur Fortran-S [23], par exemple, permet la parallélisation par distribution du contrôle de programmes Fortran en s'appuyant sur le système de mémoire virtuelle partagée Koan [87, 86].

La seconde approche n'impose aucune contrainte de ce type. L'idée de base, introduite par Callahan et Kennedy [26], est de partitionner la masse des données, et d'affecter chaque partition à un processeur de la machine cible. Chaque processeur exécute alors le même programme (correspondant au programme utilisateur initial), mais ne traite que les données de la partition — ou des partitions — dont il est propriétaire. On préserve ainsi la simplicité de la programmation séquentielle : le programmeur d'application spécifie la politique de distribution qu'il désire voir appliquer aux données manipulées mais n'a pas à gérer explicitement cette distribution, ni le parallélisme qui en résulte.

L'approche de la parallélisation basée sur la distribution des données est celle retenue dans les travaux qui visent au développement de compilateurs-paralléliseurs semi-automatiques pour langages de type HPF (Fortran90 « étendu » par l'adjonc-

tion de directives de distribution, d'alignement, et de placement des données [74]). Elle a déjà mené à l'élaboration de compilateurs prototypes tels que SUPERB [15], Pandore [35, 9], ou Fortran-D [113], et des outils de ce type devraient être commercialisés dans un proche avenir.

Cependant, le développement de compilateurs-parallèliseurs capables de générer du code efficace à partir d'un langage tel que HPF s'est avéré beaucoup plus difficile que prévu initialement. Bien que diverses techniques soient connues pour optimiser les programmes parallèles (vectorisation des échanges de données, réduction des domaines d'itération, etc. [117]), il est difficile d'obtenir qu'elles soient appliquées automatiquement par un compilateur, en dehors de quelques cas bien identifiés. En outre, la plupart des travaux en cours portent sur la distribution de structures de données régulières de type tableau et la parallélisation de nids de boucles affines. Les techniques d'optimisation à la compilation nécessitent par ailleurs que la distribution des données ainsi que les schémas d'accès à ces données soient connus statiquement. C'est d'ailleurs la raison pour laquelle les recherches effectuées jusqu'à ce jour ont été principalement concentrées sur la parallélisation de langages réguliers tels que le sous-ensemble « statique » de HPF.

Des propositions s'appuyant sur des techniques dérivées du domaine de l'intelligence artificielle ont été faites pour surmonter ces problèmes (*e.g.* [48]). Partant du constat que les opérations élémentaires effectuées au cœur des nids de boucles sont toutes plus ou moins conformes à quelques schémas de base qu'il est possible d'identifier une fois pour toutes, il a été proposé dans PARAMAT [39] d'identifier ces opérations de base afin de les substituer automatiquement par des opérations équivalentes, mais parallèles et optimisées.

Les approches évoquées ci-dessus présentent l'avantage de permettre une transition en douceur de programmes séquentiels pré-existants vers des programmes parallèles équivalents pour APMD. Dans le meilleur des cas il devrait suffire, pour paralléliser une application écrite en Fortran à l'aide d'un compilateur-parallèliseur semi-automatique de type HPF, d'insérer des directives de distribution aux bons endroits dans le code source.

Bien que cette approche soit intéressante lorsqu'il faut paralléliser des applications séquentielles pré-existantes, elle ne constitue pas la manière la plus directe de développer de nouvelles applications parallèles. En fait, quand les opérations élémentaires sont identifiées au préalable et stockées dans une bibliothèque sous la forme de composants logiciels réutilisables, il devient beaucoup plus aisé de les employer directement comme « briques » élémentaires lors de la construction de nouvelles applications. La programmation par objets offre ici une solution élégante pour construire et réutiliser de telles briques logicielles. Elle permet en outre d'aborder le domaine du calcul irrégulier, caractérisé par l'emploi de structures de données

irrégulières (listes, arbres, graphes, etc.) et/ou des schémas d'accès irréguliers à ces données, alors qu'il demeure peu probable que les approches fondées sur des techniques de parallélisation à la compilation permettent d'aborder ce domaine dans un proche avenir.

Cette thèse s'inscrit dans le cadre du projet EPEE (Environnement Parallèle d'Exécution de Eiffel), dont l'objectif est de promouvoir l'idée que les mécanismes de la programmation par objets peuvent contribuer à simplifier la programmation des APMD. L'environnement EPEE constitue un cadre conceptuel pour le développement de composants logiciels parallèles portables, flexibles et performants. Dans les contributions de cette thèse, on peut distinguer :

- l'aspect méthodologique : j'ai caractérisé les objets pouvant être distribués et exploités en parallèle dans l'environnement EPEE, et proposé des schémas conceptuels permettant de développer de tels objets en insistant sur les points clés mis en avant dans les techniques modernes de génie logiciel, à savoir la maîtrise de la complexité (obtenue par la modularisation, l'encapsulation, l'héritage), la maintenabilité (corrective et évolutive), et la réutilisabilité ;

- l'illustration : j'ai développé une bibliothèque parallèle de démonstration afin de valider cette approche. Cette bibliothèque, baptisée Paladin, est dédiée au calcul d'algèbre linéaire sur APMD ;

- l'extension d'EPEE : au cours du développement de la bibliothèque Paladin, j'ai été amené à développer certains concepts génériques (*design patterns*) pour la distribution des données et la parallélisation des calculs. Des mécanismes supportant ces concepts ont depuis lors été intégrés dans la « boîte à outils » d'EPEE.

Plan du document

- Dans le chapitre 1, on présente l'environnement EPEE en tant que cadre conceptuel pour la conception par objets de composants logiciels parallèles. On décrit les mécanismes de communication, d'observation et de parallélisation intégrés à EPEE au cours de ce travail de thèse. On jette ensuite les bases d'une approche méthodique pour la conception et le développement de bibliothèques parallèles, fondée sur la notion d'agrégat polymorphe.

- Au chapitre 2, on introduit la bibliothèque de démonstration Paladin, et on montre comment les mécanismes de l'abstraction de données, de l'encapsulation et de l'héritage nous ont permis de bâtir des hiérarchies de classes décrivant des vecteurs et matrices polymorphes.

- La distribution des agrégats matrices dans Paladin est abordée au chapitre 3. On montre qu'il est possible de procéder à la distribution des matrices en décomposant le problème de leur mise en œuvre en plusieurs sous-problèmes distincts pouvant être considérés, résolus et validés séparément.

- On présente au chapitre 4 les diverses techniques permettant d'optimiser les performances globales de la bibliothèque Paladin (optimisation des calculs de localité, parallélisation des algorithmes, optimisation des communications, etc.). On montre en outre que l'optimisation d'une bibliothèque telle que Paladin peut être réalisée de manière incrémentale et maintenue transparente pour l'utilisateur.

- Dans le chapitre 5, on aborde les problèmes posés par le polymorphisme des agrégats matrices et vecteurs. On montre que la redistribution des matrices peut être obtenue aisément, et que plusieurs approches sont envisageables pour réaliser les conversions de format impliquant un changement de type dans un contexte de programmation par objets.

- Dans le chapitre 6, on présente quelques résultats expérimentaux obtenus en portant Paladin sur diverses plates-formes parallèles.

- On conclut dans le chapitre 7 en énumérant les perspectives ouvertes par ce travail. On évoque tout d'abord les possibilités offertes par la bibliothèque Paladin. On discute ensuite de la possibilité de transférer certaines des techniques dynamiques expérimentées avec Paladin dans un compilateur-parallèliseur de type HPF. On propose enfin d'aborder la distribution et le traitement en parallèle d'agrégats irréguliers en utilisant la même approche et les mêmes mécanismes que ceux qui nous ont permis de bâtir Paladin.

Guide de lecture

Dans ce document, il est fait très souvent référence à des mécanismes ou à des concepts propres au domaine de la programmation par objets (distinction entre type statique et type dynamique, mécanisme de liaison dynamique, etc.). On a introduit dans le texte de ce document de courts encadrés explicatifs, identifiés par le titre *point de langage*, à l'intention des lecteurs non familiarisés avec les techniques de programmation par objets. Il va de soi que le lecteur pour qui ces notions sont déjà familières pourra s'abstenir de lire ces encadrés, ou bien ne s'y référer qu'en cas de besoin.

Ce document contient également un grand nombre d'exemples illustrant les concepts, les mécanismes et les classes de bibliothèque évoqués au fil des chapitres.

Ces exemples sont tous exprimés à l'aide du langage Eiffel. Les constructions syntaxiques et les mécanismes particuliers à ce langage sont également décrits brièvement dans des points de langage.

Chapitre 1

L'environnement EPEE

1.1 Parallélisme et programmation par objets

1.1.1 Langages à objets parallèles

Au cours des dernières années, de nombreuses propositions ont été faites pour combiner les mécanismes de la programmation parallèle et ceux de la programmation par objets.

Le parallélisme est souvent introduit dans les langages à objets à partir de l'idée que certains objets peuvent être rendus « actifs », c'est-à-dire assimilés à des processus communiquant par échanges de messages. Les langages POOL-T [6] et ABCL/1 [121, 122, 109] sont des langages à objets parallèles mettant en œuvre des objets actifs. Une approche alternative pour introduire les notions d'activité et de concurrence dans un langage à objets consiste à maintenir les objets comme étant des entités passives pouvant être manipulées simultanément par des processus concurrents. Les langages ConcurrentSmalltalk [120, 119], DistributedSmalltalk [16] et ARCHE [17, 18] sont des langages conçus selon ce principe. Certains langages à objets parallèles, comme par exemple Eiffel // [29, 30, 31], permettent de manipuler à la fois des objets actifs et des objets passifs.

Selon les langages, les échanges de données sont réalisés par passages de messages ou par appels de méthodes à distance. La plupart des langages permettent en outre de réaliser des communications synchrones (l'objet invoquant une méthode sur un objet distant est bloqué jusqu'au terme de l'exécution de cette méthode) ou asynchrones (l'objet appelant peut poursuivre son activité en parallèle avec l'objet exécutant la méthode invoquée).

On peut encore distinguer entre les langages dans lesquels les synchronisations entre activités concurrentes sont réalisées grâce à l'emploi d'objets partagés (POOL-T, DistributedSmalltalk, Eiffel //), et ceux où l'on s'appuie sur des communications

synchronisantes (ABCL/1, ConcurrentSmalltalk).

Une étude comparative récente des langages à objets parallèles peut être trouvée dans le mémoire de thèse de C. Gransart [56]. Dans ce même document, l'auteur introduit un nouveau langage à objets parallèle baptisé BOX, qui permet au programmeur de choisir entre divers modèles de programmation (objets actifs ou objets passifs plus processus), de communication (par passages de messages ou par appels de méthodes à distance), et de synchronisation (par objets partagés ou à l'aide de communications synchronisantes).

Certains des langages à objets parallèles évoqués ci-dessus sont de *nouveaux* langages, conçus spécialement afin de permettre la programmation d'applications parallèles à l'aide d'objets répartis (c'est par exemple le cas des langages POOL-T, ABCL/1, et BOX). Les autres langages sont obtenus par *extension* d'un langage à objets séquentiel pré-existant : les langages ConcurrentSmalltalk et Distributed-Smalltalk sont ainsi des extensions du langage séquentiel Smalltalk [53], de même que COOL [33] est une extension de C++, que Eiffel // étend le langage Eiffel, et que pSather [99] étend le langage Sather [101]. En général, le parallélisme est introduit dans les langages de ce type grâce à l'emploi de directives de compilation, ou encore par l'insertion de macro-instructions dans les classes d'application.

La mise en œuvre de tels langages — qu'il s'agisse de nouveaux langages ou de langages étendus — implique le développement d'un compilateur *ad hoc*, ou tout au moins celui d'un pré-processeur.

Certains auteurs ont proposé une autre manière d'introduire le parallélisme dans un langage à objets. L'idée est d'encapsuler des mécanismes parallèles dans des classes d'un langage à objets purement séquentiel. Ces classes peuvent ensuite être réutilisées grâce au mécanisme de l'héritage pour construire des objets actifs, sans qu'il faille pour cela modifier la syntaxe du langage utilisé, ni développer un compilateur spécialisé. Dans [38], J.-F. Colin et J.-M. Geib décrivent ainsi un ensemble de classes qui apportent au langage à objets séquentiel Eiffel des mécanismes permettant, d'une part de décrire l'activité d'un objet, et d'autre part de synchroniser des activités concurrentes. Ils montrent ensuite qu'en combinant ces mécanismes on peut mettre en œuvre des objets actifs. Dans [82], M. Karaorman et J. Bruno proposent également d'encapsuler dans des classes réutilisables des *abstractions pour le parallélisme*, comme par exemple la notion d'objet actif et celle d'invocation de méthode à distance.

1.1.2 Programmation par objets et parallélisme de données

La plupart des langages à objets parallèles évoqués dans le paragraphe précédent privilégient un modèle de programmation MIMD (*Multiple Instructions, Multiple Data*), qui est particulièrement bien adapté à la gestion d'un parallélisme de type fonctionnel : chaque tâche à exécuter est divisée en un certain nombre de sous-tâches pouvant être traitées dans des flots de calcul concurrents. Une application est donc découpée en un ensemble de processus communicants — ou d'objets actifs communicants, ce qui revient au même — dont la structure dépend de l'application considérée et de la manière selon laquelle le programmeur choisit de décomposer l'application.

Le parallélisme fonctionnel convient bien à certains types de problèmes, et notamment ceux où le parallélisme pré-existe et où les problèmes se posent essentiellement en termes de coopération. Parmi les applications de ce type, on peut citer les systèmes d'exploitation distribués et les systèmes de contrôle réparti de processus industriels.

L'emploi d'un langage à objets parallèle exige un engagement important de la part du programmeur d'application. Celui-ci doit spécifier explicitement le découpage fonctionnel de l'algorithme considéré et décrire la coopération entre les processus, tout en évitant les problèmes bien connus d'interblocage, de réception non spécifiée, etc. De plus, le mécanisme de synchronisation est difficile à hériter [96], ce qui anéantit les avantages des langages à objets eu égard à la maintenabilité du code. Enfin, les processus résultant du découpage fonctionnel étant par nature hétérogènes, des problèmes d'équilibrage de charge doivent être réglés soit par le système d'exploitation (ce qui peut être coûteux), soit par le programmeur d'application lui-même. Ceci s'avère difficile lorsque le nombre de processus disponibles sur l'architecture cible dépasse quelques dizaines d'unités, et devient pratiquement irréalisable lorsque l'architecture offre plusieurs centaines de processeurs.

Pour surmonter ces obstacles, nous proposons d'adopter un modèle de parallélisme résultant de la distribution des données, communément référencé sous le terme de « parallélisme de données » (*data parallelism*). Cette approche est aussi celle retenue dans les travaux qui visent au développement de compilateurs-paralléliseurs semi-automatiques pour langages de type HPF [124, 10, 35, 26].

Le parallélisme obtenu en distribuant les données s'inscrit tout naturellement dans un cadre de programmation par objets dans la mesure où, dans ce type de programmation, on se focalise sur les données manipulées plutôt que sur les fonctions avec lesquelles on les manipule.

Quelques principes pour construire un langage à objets gérant le parallélisme

de données ont été introduits dans [84], et les idées de bases pour étendre C++ dans le même sens ont été présentées dans [34]. Une extension de C++ a d'ailleurs été effectivement réalisée avec le langage pC++ [22], et une extension de Sather a été proposée dans [108]. Dans tous ces cas, cependant, il s'agit de développer un langage à objets doté de constructions *explicites* pour le parallélisme de données, ce qui implique le développement d'un compilateur — ou tout au moins d'un pré-processeur — approprié. Avec EPEE, notre Environnement Parallèle d'Exécution de Eiffel, nous proposons d'intégrer totalement la distribution de données et le parallélisme résultant de cette distribution dans un langage à objets séquentiel, sans qu'aucune modification soit apportée à la syntaxe ou à la sémantique de ce langage, et sans qu'il faille développer de nouveau compilateur. Notre approche s'inscrit donc dans la même optique que celles visant à encapsuler dans des classes des *abstractions pour la programmation parallèle*, sans altération du langage séquentiel utilisé [38, 82, 42]. Dans notre cas, cependant, l'objectif n'est pas de construire des objets *actifs*, mais de masquer à l'utilisateur la distribution des données et leur traitement en parallèle.

1.2 L'environnement EPEE

L'acronyme EPEE désigne un Environnement Parallèle d'Exécution de Eiffel, qui a vu le jour en 1991 à l'IRISA à l'initiative de Jean-Marc Jézéquel [80]. Cet environnement constitue un cadre conceptuel[1] pour le développement de composants logiciels réutilisables pouvant servir à la programmation des architectures parallèles à mémoire distribuée (APMD).

L'originalité du projet EPEE vient de ce qu'on utilise un langage à objets purement séquentiel, le langage Eiffel, pour encapsuler dans des classes des mécanismes de distribution des données et de parallélisation des calculs, en adoptant un modèle d'exécution SPMD qui permet de décomposer une application en un entrelacement de phases parallèles et de phases séquentielles, conformément au modèle BSP (*Block Synchronous Parallel*) introduit par Leslie G. Valiant dans [114]. Cette approche permet de construire dans un contexte de génie logiciel des bibliothèques souples et évolutives présentant des interfaces séquentielles à leurs utilisateurs, tout en ayant des réalisations parallèles efficaces [62, 63].

Le parallélisme de données est particulièrement bien adapté aux algorithmes manipulant des *agrégats*, c'est-à-dire de très grandes structures de données homogènes. L'environnement EPEE permet de développer des composants logiciels réutilisables

1. On commence aujourd'hui à employer le terme de « *design framework* » pour désigner ce type d'environnement [52, 45].

décrivant, d'une part des agrégats pouvant être distribués sur une APMD, d'autre part des algorithmes parallèles manipulant ces agrégats de manière efficace. En utilisant le mécanisme de l'encapsulation pour masquer à l'utilisateur les détails de la mise en œuvre d'un objet (principe du masquage d'information), la distribution des agrégats et le parallélisme afférant peuvent être intégrés proprement dans des classes, sans qu'il soit nécessaire d'apporter la moindre modification à la syntaxe ou à la sémantique du langage séquentiel utilisé.

1.2.1 Travaux connexes

La plupart des travaux visant à encapsuler le parallélisme de données dans les classes d'un langage à objets séquentiel s'appuient sur le langage C++, et se limitent souvent à la seule distribution des objets de type tableau. On évoque ci-dessous quelques uns de ces travaux, choisis parmi les plus récents.

- Dans [91], les auteurs proposent d'introduire le parallélisme de données à l'aide des *templates* de C++, en s'inspirant du modèle de *C** [51], mais en conservant la syntaxe de C++. L'approche proposée implique cependant qu'un pré-processeur soit ensuite utilisé pour « traduire » le code C++ data-parallèle obtenu en code *C**.

- La bibliothèque P++ [89] met en œuvre des *grilles virtuelles partagées*. Elle a été conçue afin de servir de support à la mise en œuvre de méthodes de résolution adaptatives parallèles pour la modélisation des phénomènes de combustion. P++ s'appuie sur le langage C++, mais fait intervenir les classes séquentielles de manipulation de tableaux de la bibliothèque commerciale M++.

- L'environnement PARLANCE (*Parallel Library and Networked Computing Environment* [4, 5]) fournit une interface C++ pour le développement et l'utilisation de bibliothèques d'algorithmes parallèles depuis des applications exprimées en Fortran90. PARLANCE permet la distribution de tableaux multidimensionnels (selon des schémas de distribution inspirés de HPF), et offre des *squelettes* de communication et de calcul pour exploiter ces tableaux distribués. Les mécanismes mis en œuvre dans PARLANCE ont pour la plupart été écrits en C++, mais présentent tous une interface Fortran qui les rend utilisables depuis des programmes exprimés en Fortran90.

- L'environnement Dome (*Distributed Object Migration Environment* [14]) est dédié au développement en C++ d'applications à objets pour réseaux de machines distribuées hétérogènes. L'objectif du projet Dome est d'offrir à l'utilisateur un modèle de programmation SPMD, les objets manipulés (par

exemple des vecteurs) étant partitionnés et distribués sur les machines du réseau. L'originalité du projet Dome est que chaque partition n'est pas attachée à une machine particulière mais peut migrer dynamiquement d'une machine à l'autre en fonction de la charge du réseau. Le projet Dome ne cible donc pas les super-calculateurs parallèles, mais est plutôt destiné à la mise en œuvre d'applications SPMD pour réseaux de stations de travail.

1.2.2 Choix du langage

L'environnement EPEE s'articule, comme son nom l'indique, autour du langage à objets à typage statique Eiffel. Ce langage a été conçu par Bertrand Meyer en 1985. Plusieurs compilateurs de souches différentes sont maintenant disponibles pour ce langage, et commercialisés par la société SIG en Allemagne et par les sociétés ISE[2] et Tower Technology aux États-Unis.

Le langage Eiffel a été choisi pour servir de support à l'environnement EPEE parce qu'il s'agit d'un langage à objets moderne, à la sémantique clairement définie, et offrant tous les concepts dont nous avons besoin. Cependant, les idées fondamentales mises en avant dans le cadre du projet EPEE ne sont aucunement dépendantes de ce langage. Tout autre langage à objets offrant l'encapsulation stricte, l'héritage, la liaison dynamique, le polymorphisme de référence et une certaine forme de généricité pourrait être utilisé à la place d'Eiffel. Les langages C++, Modula 3 et Ada 95, par exemple, pourraient ainsi servir au développement de bibliothèques d'agrégats distribués semblables à celles que nous développons avec EPEE.

1.2.3 Description de l'environnement

L'environnement EPEE peut en fait être perçu comme une sorte de « boîte à outils » logicielle comprenant notamment :

- des outils de compilation croisée permettant de générer à partir de code Eiffel du code exécutable pour, potentiellement, n'importe quelle APMD cible ;
- un ensemble de classes Eiffel qui fournissent au programmeur des mécanismes génériques (*design patterns*) grâce auxquels il lui est possible de gérer la distribution des données et leur manipulation en parallèle.

Une première maquette de l'environnement EPEE, construite en 1991 sur la base du langage Eiffel 2 [98], a permis de montrer qu'il est effectivement possible d'intégrer totalement la distribution des données et leur traitement en parallèle dans les classes d'un langage à objets séquentiel [80]. Depuis lors, l'environnement EPEE

2. ISE : *Interactive Software Engineering.*

a été adapté au langage Eiffel 3 [97] et mis en œuvre sur réseau de stations de travail, sur la machine Intel iPSC/2 et sur la machine Intel Paragon XP/S.

Le travail de thèse rapporté dans ce document s'est inscrit dans le cadre du développement et de l'expérimentation d'EPEE. Au cours de ce travail, j'ai notamment été amené à intégrer dans l'environnement EPEE une nouvelle bibliothèque de communication et d'observation conçue dans le cadre du projet Pampa. L'intégration de cette bibliothèque dans l'environnement EPEE fait l'objet du paragraphe 1.2.4. J'ai également conçu et intégré à la « boîte à outils » d'EPEE un certain nombre de mécanismes génériques puissants pour aider à la distribution des données et à la parallélisation des calculs dans les applications SPMD. Ces mécanismes sont évoqués dans le paragraphe 1.2.5, et l'un d'entre eux est décrit en détails dans l'annexe B (page 225).

1.2.4 Support de communication et d'observation : la POM

1.2.4.1 Description

La machine virtuelle POM (*Portable Observable Machine*) présente une interface système homogène capable de masquer les caractéristiques architecturales d'un grand nombre de machines parallèles et distribuées. Elle a été conçue afin de servir de support commun aux divers environnements de programmation développés dans le cadre du projet Pampa, à savoir le compilateur-paralléliseur Pandore [8], l'environnement dédié au prototypage d'algorithmes parallèles ECHIDNA [77], et l'environnement EPEE.

La POM permet en outre la connexion de ces environnements de programmation avec des outils de collecte et d'analyse de traces d'exécution. Ceux-ci ont pour fonction d'aider à la mise au point des programmes sur diverses architectures en fournissant au concepteur les indices nécessaires à la compréhension du comportement de son programme. Parmi les mécanismes d'observation intégrés à la POM, on peut citer l'estampillage des événements, à l'aide d'estampilles vectorielles ou d'estampilles adaptatives [78]. L'estampillage permet l'analyse des synchronisations qui se produisent entre les nœuds d'application à l'exécution [76]. La POM peut aussi assurer la datation physique des événements à partir d'un temps global reconstitué après évaluation des dérives des horloges locales aux nœuds de la machine parallèle cible[3]. Elle permet également le déport de l'analyse sur un nœud observateur (voir figure 1.1), la machine virtuelle assurant l'acheminement des messages de trace vers

3. L'idée de base est exposée dans [70]. Elle a été pour la première fois implantée sur machine parallèle pour Echidna [79]. Une variante a été développée dans l'équipe Apache à Grenoble [95], et intégrée ensuite à la POM.

le nœud observateur en essayant de perturber le moins possible le fonctionnement du programme sous test.

Au cours de l'année universitaire 1993–1994, j'ai participé activement à la spécification de la bibliothèque POM, qui a ensuite été implantée sur plusieurs plates-formes et systèmes d'exploitation par divers membres de l'équipe Pampa. À ce jour la bibliothèque a été portée sur les machines Intel iPSC/2 et Paragon XP/S, sur réseau de stations de travail (communiquant par câble Ethernet ou par liaison ATM), et permet également la simulation d'une machine parallèle sur une seule station de travail. J'ai en outre réalisé la mise en œuvre de la bibliothèque au dessus de PVM [110].

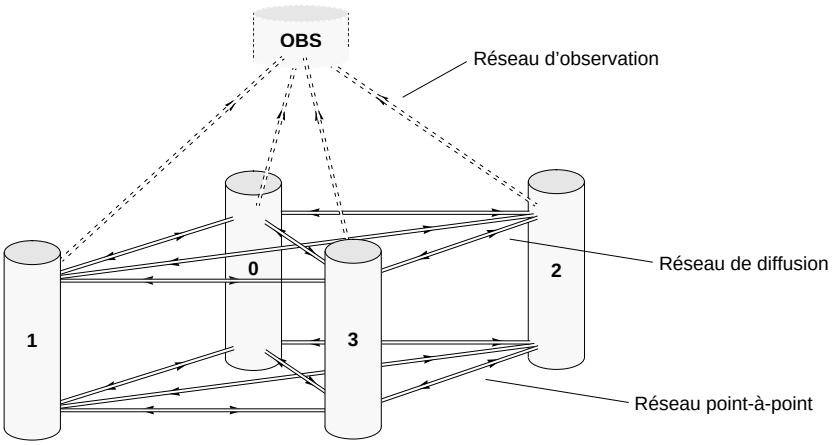


FIG. 1.1 - *Modèle de la machine virtuelle POM*

1.2.4.2 Intégration de la POM dans le monde Eiffel

Plusieurs classes Eiffel ont été développées afin d'assurer l'interface entre la bibliothèque POM (écrite en C) et le monde Eiffel. Ces classes font à présent partie intégrante de la « boîte à outils » d'EPEE. Grâce à elles, toute bibliothèque Eiffel construite avec EPEE peut être aisément portée sur n'importe quelle machine parallèle cible, pourvu que la bibliothèque POM y ait été portée au préalable.

La classe POM reproduit au niveau du langage Eiffel tous les services de communication et d'observation offerts au niveau du langage C par la bibliothèque POM. L'interface de cette classe est partiellement reproduite dans l'exemple 1.1.

La plupart des services de communication et d'observation offerts par cette classe sont directement calqués sur ceux de la bibliothèque POM. Cependant, la

Exemple 1.1

```
expanded class interface POM
    -- Communication and observation features offered by the POM library

feature -- General purpose features
    node_id : INTEGER                                                    5
            -- Identity of local node
    nb_nodes : INTEGER
            -- Number of nodes in the application

feature -- Emission                                                     10
    send (pid : INTEGER ; object : ANY)
            -- Send object to node 'pid'
    bcast (object : ANY)
            -- Broadcast object to all nodes
                                                                        15
feature -- Reception
    recv_from (pid : INTEGER ; object : ANY)
            -- Receive object from node 'pid' (point-to-point mode)
    recv_bcast_from (pid : INTEGER ; object : ANY)
            -- Receive object from node 'pid' (broadcast mode)         20

feature -- Test
    probe_from (pid : INTEGER) : INTEGER
            -- Test for object arriving from node 'pid' (point-to-point mode)
    probe_bcast_from (pid : INTEGER) : INTEGER                          25
            -- Test for object arriving from node 'pid' (broadcast mode)

feature -- Trace management
    trace (name : STRING ; data_length : INTEGER ; data : POINTER)
            -- Send a trace message to the observation node.            30
    trace_on
            -- Enable automatic tracing of communications
    trace_off
            -- Disable automatic tracing of communications
    ...                                                                 35
end -- interface POM
```

classe POM propose également un certain nombre de services qui n'ont pas d'équivalent dans la bibliothèque POM. Ainsi, on dispose par exemple d'un mécanisme permettant de tracer automatiquement tous les échanges de données réalisés entre les nœuds d'application. Ce mécanisme peut être activé ou désactivé à volonté en invoquant les routines *trace_on* et *trace_off* de la classe POM.

Sémantique des « transmissions d'objets »

Il est important de noter que dans le schéma de communication qui nous intéresse dans EPEE, un objet ne peut en aucun cas « migrer » entre les nœuds d'application comme c'est le cas dans certains systèmes à objets distribués [14]. Seules les données encapsulées au sein d'un objet peuvent être émises vers un ou plusieurs nœuds distants. Au niveau des nœuds destinataires, il faut impérativement qu'un objet de même type que celui de l'objet émis soit disponible pour qu'on y place les données reçues. C'est d'ailleurs pour cette raison raison que les routines de réception de la classe POM sont des procédures prenant en paramètre un objet destiné à servir de « réceptacle » aux données reçues, et non des fonctions capables de créer un nouvel objet et de l'initialiser avec les données reçues avant de le retourner en résultat.

Ce schéma peut paraître restrictif, mais il convient en fait parfaitement au modèle d'exécution SPMD qui nous intéresse dans le cadre du projet EPEE. En effet, lorsqu'un nœud quelconque doit émettre un objet vers un ou plusieurs nœuds destinataires, ces derniers connaissent très précisément la nature des données à recevoir puisqu'ils exécutent le même programme d'application. Les échanges de données étant déterministes dans le contexte qui nous intéresse, on peut toujours faire en sorte qu'au niveau des nœuds destinataires un objet ayant le type requis serve de réceptacle aux données reçues.

Il serait parfaitement envisageable de modifier la classe POM afin d'autoriser les communications non-déterministes. Il faudrait pour cela associer aux données émises des informations concernant le type de l'objet transmis. Chaque nœud destinataire pourrait alors créer localement un objet ayant le type indiqué avant d'y placer les données reçues.

Caractérisation des objets transmissibles

Dans le domaine de la programmation par objets, on a coutume de s'intéresser en priorité aux propriétés des objets plutôt qu'à celles des programmes ou des algorithmes. Nous avons donc développé une classe afin de faire bénéficier les objets Eiffel des services de communication offerts par la POM : la classe TRANSMISSIBLE caractérise les objets dont le contenu peut être transmis entre les nœuds d'une APMD. Son interface est reproduite dans l'exemple 1.2.

Exemple 1.2

```
class interface TRANSMISSIBLE

feature -- Transmission features
    send (destination : INTEGER)
            -- Send object to 'destination'                          5
    bcast
            -- Broadcast object to all nodes
    recv_from (source : INTEGER)
            -- Receive object from 'source' in point-to-point mode
    recv_bcast_from (source : INTEGER)                               10
            -- Receive object from 'source' in broadcast mode

end -- interface TRANSMISSIBLE
```

Grâce au mécanisme de l'héritage, toute classe héritant de la classe TRANSMISSIBLE décrit à son tour des objets transmissibles.

1.2.5 Des mécanismes génériques pour la parallélisation

En utilisant les mécanismes de communication élémentaires offerts par la POM, on peut développer et encapsuler dans des classes des mécanismes puissants susceptibles d'aider à la distribution des données, aux mouvements de données (mécanismes de diffusion sélective, de transfert par décalage, etc.), ou à la parallélisation des calculs (opérations de type *scatter/gather*, *apply/reduce*, etc.). Les classes encapsulant de tels mécanismes constituent des *abstractions parallèles*. Elles peuvent être développées au cas par cas en fonction des besoins, et intégrées alors dans la boîte à outils d'EPEE qui se trouve ainsi enrichie de manière incrémentale.

Jusqu'à présent, nous avons intégré à la boîte à outils d'EPEE des mécanismes capables de gérer la distribution « à la HPF » de structures de données mono- et bi-dimensionnelles de type tableau. Ces mécanismes seront présentés en détails dans le chapitre 3.

Nous avons également intégré à la boîte à outils des mécanismes permettant de réaliser des opérations de type *apply/reduce* dans un contexte d'exécution SPMD. L'un de ces mécanismes est décrit à titre d'exemple en annexe B (page 225).

1.3 Développement de bibliothèques parallèles avec EPEE

L'environnement EPEE constitue un cadre conceptuel pour la conception et le développement de structures de données *agrégats* pouvant être distribuées et manipulées en parallèle sur des APMD. On distinguera donc entre le concepteur de classes décrivant des agrégats, et l'utilisateur créant et manipulant de tels agrégats dans le cadre de programmes d'application SPMD. L'un des objectifs du travail de thèse rapporté dans ce document était de définir un ensemble de règles méthodologiques susceptibles de guider le concepteur de nouvelles classes d'agrégats. Dans cette optique, j'ai introduit la notion d'*agrégat polymorphe* et proposé une méthode pour la conception et le développement de bibliothèques d'agrégats polymorphes, fondée sur le principe de l'abstraction de données et s'appuyant sur les mécanismes de l'encapsulation, de l'héritage, du polymorphisme de référence et de la liaison dynamique.

Dans ce paragraphe, on montre qu'en utilisant EPEE il est possible de développer des bibliothèques d'agrégats pour APMD tout en assurant la transparence de leur mise en œuvre, leur portabilité, leur efficacité, et l'extensibilité de leurs performances.

1.3.1 Problématique

Les rares bibliothèques offertes à ce jour aux programmeurs d'APMD sont pour la plupart des bibliothèques numériques « clés en main ». Elles sont en général conçues par des experts et mises en œuvre à très bas niveau afin d'exploiter au mieux les caractéristiques architecturales de telle ou telle machine cible et d'offrir des performances optimales sur cette machine. Cette approche est très révélatrice du fait que, du point de vue des concepteurs de bibliothèques (mais aussi de certains utilisateurs), la performance est bien souvent perçue comme le *seul* objectif digne d'être considéré lorsqu'il faut juger de la qualité d'une bibliothèque. Tout se passe donc comme si les autres critères pouvant être pris en compte — comme par exemple la portabilité de la bibliothèque, sa facilité d'emploi, sa flexibilité, etc. — devaient nécessairement s'effacer devant la performance. Pourtant, le portage d'une bibliothèque dont les routines sont étroitement dépendantes des caractéristiques des plates-formes cibles demande un effort considérable, dont on peut se demander s'il est rentable en termes de génie logiciel, eu égard à la courte durée de vie des machines parallèles.

Le domaine du calcul numérique, et plus particulièrement celui du calcul d'algèbre linéaire, est certainement celui dans lequel cette prédominance du « critère performance » est la plus marquée. Des routines permettant de réaliser les opérations

élémentaires telles que les opérations vecteur-vecteur, vecteur-matrice et matrice-matrice sont en général mises en œuvre par les fabriquants des machines parallèles eux-mêmes. Il en va du prestige de ces fabriquants et de la notoriété de leur machine, puisque c'est très souvent sur la base des performances observées en exécutant des algorithmes d'algèbre linéaire types (produit de matrices, factorisation LU, etc.) que cette machine va être jugée. Cette dictature du critère performance peut d'ailleurs entraîner très loin les fabriquants attachés à montrer que leur architecture parallèle est « la meilleure » : la mise en œuvre de la routine calculant le produit de matrices dans la bibliothèque scientifique de la CM-2 (*Connection Machine*) a nécessité approximativement 10 hommes-années d'efforts [41].

La bibliothèque ScaLAPACK [36] fournit au programmeur numéricien un ensemble de routines Fortran pré-définies avec lesquelles il peut développer des programmes d'application SPMD pour machines à mémoire distribuée, tout en s'affranchissant des problèmes de parallélisation des calculs : les algorithmes parallèles sont déjà encapsulés dans la bibliothèque. ScaLAPACK est bâtie au dessus de la bibliothèque LAPACK [7], qui fait depuis longtemps figure de référence auprès des programmeurs numériciens développant des programmes Fortran pour machines séquentielles.

LAPACK et ScaLAPACK sont toutes deux mises en œuvre de manière à fournir les meilleures performances possibles. En revanche, elles demeurent totalement fermées à toute espèce d'extension de la part de l'utilisateur, c'est-à-dire qu'elles lui interdisent toute intervention au niveau de leur structure ou de leur code. Un programmeur numéricien qui désirerait, par exemple, développer des programmes procédant à des calculs d'algèbre linéaire sur des vecteurs et matrices de nombres rationnels ne pourrait utiliser à cette fin ni LAPACK, ni ScaLAPACK, aucune de ces deux bibliothèques n'étant générique. Ce programmeur n'aurait donc d'autre ressort que de réécrire intégralement une nouvelle bibliothèque de routines adaptée à ses besoins propres. Il en irait de même s'il souhaitait procéder à des calculs impliquant la manipulation de matrices et de vecteurs creux : les deux bibliothèques ont été explicitement conçues afin de ne manipuler que des vecteurs et matrices senses représentés sous la forme de tableaux Fortran denses. Il en irait encore de même si un utilisateur de ScaLAPACK voulait expérimenter ses propres schémas de distribution plutôt que de s'appuyer sur les seuls schémas de distribution par blocs rectangulaires (*SBS : Square Block Scattered*) admis par les routines de cette bibliothèque.

Les bibliothèques « clés en mains » telles que LAPACK et ScaLAPACK sont donc conçues de manière à offrir les meilleurs performances possibles, mais demeurent fermées à toute espèce d'extension de la part de l'utilisateur. Avec l'envi-

ronnement EPEE, nous proposons une nouvelle manière d'aborder le processus de développement des bibliothèques parallèles pour APMD.

Notre approche s'appuie sur les mécanismes fondamentaux de la programmation par objets (abstraction de données, principe d'ouverture/fermeture des classes, modularité, encapsulation, héritage, liaison dynamique, etc.). Nous voulons montrer qu'à l'aide de ces mécanismes, il est possible de développer des bibliothèques pour APMD qui soient à la fois modulaires, extensibles, flexibles, d'un emploi aisé, et néanmoins performantes. Les résultats attendus de cette manière de concevoir le développement de bibliothèques sont concomittants de l'approche orientée objet : qualité du logiciel, réutilisabilité, réduction des temps et coûts de développements, de mise au point et de maintenance.

1.3.2 Objectifs de qualité

Dans ce paragraphe sont énumérés les critères de qualité qu'il nous paraît souhaitable de chercher à privilégier lors du développement de bibliothèques parallèles. Pour chacun de ces critères, on précise quels sont les atouts de l'environnement EPEE ou de la programmation par objets.

1.3.2.1 Portabilité

Les bibliothèques parallèles conçues dans l'environnement EPEE doivent être aisément portables sur un grand nombre de plates-formes parallèles de type APMD, et notamment pouvoir s'adapter rapidement à de nouvelles architectures. Pour ce faire, il importe que le code source des bibliothèques soit lui-même portable, et aussi peu dépendant que possible des caractéristiques physiques de l'architecture sous-jacente.

- **Portabilité du code source**
  Dans l'environnement EPEE, la portabilité du code source est due au fait que les compilateurs commerciaux actuels pour le langage Eiffel (compilateurs commercialisés par les sociétés ISE, Tower Technology, et SIG) utilisent tous le langage C comme langage intermédiaire. Les environnements logiciels des APMD intégrant tous au moins un compilateur C, les applications et composants logiciels développés dans le cadre d'EPEE sont donc aisément portables sur ces machines.

- **Indépendance vis à vis des architectures**
  L'indépendance des applications développées avec EPEE vis à vis des caractéristiques architecturales des diverses APMD résulte de l'interfaçage d'EPEE

avec la bibliothèque de communication et d'observation POM[4]. Cette bibliothèque a été conçue afin d'être aisément portable, et offre l'abstraction d'une machine parallèle homogène. Les composants logiciels développés avec EPEE peuvent donc être construits sur la base de cette machine virtuelle. Ils bénéficient des mécanismes de communication et d'observation offerts par la bibliothèque POM et peuvent dès lors être portés et utilisés sur n'importe quelle APMD (pourvu que la POM y ait été portée au préalable).

1.3.2.2 Évolutivité

Le critère d'évolutivité des bibliothèques parallèle s'inscrit dans le cadre de la maintenance évolutive de ces bibliothèques. Il est aujourd'hui communément admis que le coût de la maintenance (corrective et évolutive) d'un projet de développement logiciel peut représenter de 2 à 5 fois le coût du développement initial. Il est donc capital de s'attacher à développer des composants logiciels pouvant être réutilisés et/ou étendus à volonté. Les techniques de programmation par objets sont particulièrement bien adaptées à ce type de développement.

Les bibliothèques développées avec EPEE doivent demeurer extensibles à tout instant, ce qui signifie qu'on doit pouvoir y encapsuler à volonté de nouvelles routines pour manipuler les agrégats considérés (extensibilité algorithmique), ou de nouvelles classes décrivant des mises en œuvres alternatives pour ces agrégats (polymorphisme).

- **Extension algorithmique**
  En programmation par objets, il est toujours possible d'incorporer de nouvelles routines dans une classe, ou de redéfinir une routine héritée d'une classe parente. On montrera dans les chapitres suivants comment on peut définir au niveau d'une classe encapsulant la spécification « abstraite » d'un agrégat des routines permettant de procéder à des opérations avec cet agrégat, et comment on peut ensuite redéfinir ces routines dans des classes héritant de la classe abstraite afin de les optimiser et/ou de leur donner une mise en œuvre parallèle.

- **Polymorphisme**
  Il est toujours possible d'incorporer dans une hiérarchie de classes existante de nouvelles classes en s'appuyant sur le mécanisme de l'héritage. On montrera dans les chapitres suivants comment, partant d'une classe encapsulant la spécification « abstraite » d'un agrégat, on peut développer peu à peu toute une hiérarchie de classes héritant de la classe abstraite et décrivant de nouveaux formats de représentation interne pour ce type d'agrégat.

4. La bibliothèque POM a été évoquée au § 1.2.4.

1.3.2.3 Performance

La programmation par objets amène au développement de hiérarchies de classes complexes. Il n'est donc pas étonnant que l'on s'inquiète de l'efficacité de leur mise en œuvre. L'efficacité est en fait un problème crucial dans le cas du développement de bibliothèques d'agrégats pour APMD, puisque notre principale motivation est de pouvoir exploiter l'impressionnante puissance de calcul de ces machines. Si on devait aboutir à des programmes d'application SPMD s'exécutant plus lentement sur une APMD que la (meilleure) application séquentielle équivalente, on aurait évidemment travaillé en pure perte. Fort heureusement, tel n'est pas le cas.

Dans l'environnement EPEE nous bénéficions du fait que, Eiffel étant un langage très récent, les compilateurs développés pour ce langage mettent en œuvre les techniques de compilation les plus modernes.

Optimisations réalisées à la compilation

La grande modularité permise dans les langages à objets est souvent perçue comme une barrière à toute espèce d'optimisation du code généré. En fait, grâce à la sémantique claire du langage, les compilateurs Eiffel actuels sont capables de réaliser des optimisations très avancées en procédant à une analyse globale du système en cours de compilation (*system wide analysis*). Ils peuvent ainsi détecter les routines pouvant être invoquées statiquement, et remplacer alors la liaison dynamique par un simple appel de procédure. Le mécanisme de la liaison dynamique n'intervient donc que lorsque le compilateur est incapable d'identifier statiquement quelle version d'une routine doit être exécutée. Ce problème se pose seulement lorsque la routine considérée a été définie plusieurs fois dans une hiérarchie de classes et que la connaissance du type de base de l'objet sur lequel la routine doit être invoquée ne permet pas de réduire l'ensemble des implantations candidates à un seul élément.

Les compilateurs actuels peuvent aussi éviter le coût de certains appels de procédure en procédant à des expansions de code (*inline expansion*), et en supprimant du code généré les routines qui ne sont jamais invoquées et les segments de code qui demeurent inutilisés (*dead code removal*). Les compilateurs Eiffel actuels réalisent tous de telles optimisations automatiquement. Une bonne partie du surcoût lié à la modularité est donc éliminée sans que l'intégrité du système compilé soit compromise.

Dès lors qu'un compilateur Eiffel a produit un code intermédiaire (qui s'avère être un code C standard avec les environnements Eiffel commercialisés actuellement), les méthodes d'optimisation habituelles peuvent être appliquées par un compilateur traditionnel. Ce compilateur procède aux opérations de fusion de boucles (*loop merging*) et de déroulage de boucles (*loop unrolling*), à l'élimination des sous-expressions redondantes, etc.

Ramasse-miettes automatiques

Les ramasse-miettes automatiques tels que ceux qui sont intégrés aux environnements Eiffel sont souvent soupçonnés d'être trop coûteux. Cette mauvaise réputation remonte sans doute à l'apparition des premiers systèmes Lisp, dont les ramasse-miettes assuraient la gestion de la mémoire d'une manière très inefficace. Pourtant, les mécanismes de gestion automatique de la mémoire ont énormément progressé depuis lors, et des expériences récentes [85, 54, 116] ont montré qu'utiliser un ramasse-miettes moderne n'est pas plus coûteux que de réaliser la gestion de la mémoire « manuellement » (en utilisant par exemple les fonctions C traditionnelles *malloc* et *free*). Par ailleurs, on montrera dans le paragraphe 4.3.3 que, dans l'environnement EPEE, on peut faire en sorte que les périodes d'activité du ramasse-miettes recouvrent les phases de communication.

Contrôle assertionnel

Le langage Eiffel permet d'associer à chaque routine une précondition et une postcondition, de spécifier des invariants de classe, etc. Les assertions de ce type jouent un rôle majeur dans le langage Eiffel en aidant à développer des classes correctes et auto-documentées (c'est-à-dire des classes dont l'interface et les propriétés essentielles peuvent être extraites automatiquement par des outils appropriés). Elles apportent en outre un confort indéniable lors du débogage, chaque clause assertionnelle non respectée déclenchant immédiatement la génération d'une exception.

Cependant, lorsqu'on invoque à l'exécution les routines d'une classe comportant de nombreuses clauses assertionnelles, les performances de l'application globale peuvent s'en trouver fortement diminuées. Pour éviter ce problème, il est possible de désactiver le contrôle assertionnel sur tout ou partie des classes d'un système lors de sa compilation. Les classes pour lesquelles on désactive le contrôle assertionnel doivent bien sûr être des classes en lesquelles on a toute confiance, c'est-à-dire des classes qui ont déjà été maintes fois testées. En règle générale, on désactivera le contrôle assertionnel au moins pour les classes de la bibliothèque standard d'Eiffel.

Performance du code exécutable généré par les compilateurs Eiffel

Lorsque le travail de thèse rapporté dans ce document a débuté en 1993, les performances du code généré par le compilateur Eiffel d'ISE dont nous disposions à l'époque (version 2.3) demeuraient sans commune mesure avec celles que l'on pouvait obtenir en codant directement des programmes équivalents en C. Dans le cas de programmes manipulant des tableaux, on observait parfois une différence dans les temps d'exécution atteignant le facteur 20 (les programmes Eiffel étaient 20 fois plus lents que des programmes C équivalents).

Au fil des versions successives des compilateurs, cette différence a décru peu à peu pour atteindre le facteur 2 en moyenne avec le compilateur d'ISE version 3.2.3 qui nous a servi à réaliser la plupart des expérimentations dont les résultats sont rapportés dans le chapitre 6.

Nous n'avons malheureusement pas été en mesure de procéder aux mêmes expérimentations avec l'un des tout derniers compilateurs proposés par les sociétés Tower et ISE. Quelques expériences nous ont toutefois permis de vérifier qu'avec le dernier compilateur TowerEiffel (version 1.4.3), on atteint le facteur 1.1 : la durée d'exécution d'un programme Eiffel manipulant des tableaux de manière intensive est donc seulement 10 % plus longue que celle d'un programme C équivalent (et optimisé à la main). Par ailleurs, la société ISE a annoncé que son nouveau compilateur (version 3.3) présente des performances à peu près similaires : d'après B. Meyer, la durée d'exécution d'un programme de test manipulant des tableaux est, dans le meilleur des cas, la même que celle d'un programme C équivalent (sur station de travail DEC Alpha) et n'est que 8 % plus longue sur une station de travail Sun Sparcstation.

Avec les compilateurs Eiffel les plus récents, les performances du code objet généré sont donc tout à fait comparables avec celles d'un programme écrit en C, et on peut s'attendre à ce que ces performances augmentent encore dans un futur proche. Il faut d'ailleurs garder à l'esprit que les expériences évoquées ci-dessus portaient sur des programmes manipulant des tableaux, c'est-à-dire des structures de données extrèmement régulières dont les accès bénéficient d'un traitement *très* privilégié de la part des compilateurs C, alors que le type tableau ne fait pas partie des types prédéfinis du langage Eiffel (seuls sont prédéfinis les types de base tels que entiers, réels, booléens, etc.).

Quoiqu'il en soit, les performances observées avec les nouveaux compilateurs démontrent que, bien que la compilation de programmes à objets soit beaucoup plus difficile que celle de programmes impératifs traditionnels, on peut néanmoins atteindre des performances très honorables avec cette approche. Même s'il demeurait impossible de combler les quelques 8 à 10 % qui séparent encore les performances des programmes Eiffel de leurs homologues C, cette différence serait largement compensée par les atouts du langage Eiffel en termes de génie logiciel (typage strict, modularité, héritage, abstraction de données et encapsulation, liaison dynamique, polymorphisme de référence, ramasse-miettes, gestion des exceptions, etc.).

1.3.3 Approche méthodique

On introduit dans ce paragraphe les concepts, principes et mécanismes fondamentaux permettant de concevoir et de développer des bibliothèques d'agrégats distribués avec EPEE. Les idées énoncées ici seront illustrées dans les chapitres sui-

vants, où l'on décrira pas à pas la conception et le développement de Paladin, une bibliothèque parallèle manipulant des agrégats vecteurs et matrices distribués.

1.3.3.1 Notion d'agrégat polymorphe

La multiplicité des formats de représentation interne envisageables pour un même agrégat amène à percevoir les agrégats comme étant des entités *polymorphes*, c'est-à-dire capable d'assumer plusieurs formes et de passer dynamiquement d'une forme à l'autre.

Si l'on considère par exemple un domaine d'application dans lequel on utilise de très grandes matrices, le choix de leur format de représentation en mémoire peut être un problème crucial. En effet, bien qu'il ne doive y avoir *a priori* aucune différence entre une matrice dense et une matrice creuse du point de vue d'un programmeur d'application, la différence réside en fait dans le choix de la représentation la plus appropriée pour stocker l'une ou l'autre sorte de matrice en mémoire.

Une bonne bibliothèque d'algèbre linéaire devrait donc proposer plusieurs représentations alternatives pour les matrices et permettre au programmeur d'application de choisir — éventuellement pendant l'exécution — quelle est la représentation adéquate pour une matrice donnée. Cette bibliothèque devrait également fournir un mécanisme de conversion permettant de modifier à l'exécution la représentation courante d'une matrice : une matrice initialement creuse peut, après quelques pas de calculs, ne plus être suffisamment « creuse » pour justifier une représentation interne spéciale. On peut donc être amené à transformer une matrice creuse en une matrice dense — c'est-à-dire à modifier dynamiquement son format de représentation en mémoire — après quelques pas de calcul.

Le problème consistant à fournir plusieurs représentations alternatives d'un même agrégat se pose de manière accrue dans le contexte du calcul parallèle par distribution des données qui nous intéresse dans le cadre du projet EPEE, car les agrégats doivent être distribués sur des machines parallèles. Chaque politique de distribution envisageable pour une matrice (par lignes, par colonnes, par diagonales, par blocs, etc.) peut alors nécessiter un format de représentation interne particulier. En outre, il s'avère parfois nécessaire de redistribuer une matrice en cours d'exécution afin d'adapter sa distribution aux exigences du calcul en cours.

Le polymorphisme peut être défini comme « l'aptitude à prendre plusieurs formes ». Cette définition s'accorde bien avec les considérations précédentes sur les structures de données qui admettent plusieurs représentations en mémoire, et dont la représentation peut être modifiée dynamiquement. Partant du constat que les agrégats distribués qui nous intéressent dans le cadre du projet EPEE sont des entités poly-

morphes, je me suis attaché à proposer une méthode de développement permettant de mener à bien la mise en œuvre de tels agrégats à l'aide des services offerts par EPEE.

1.3.3.2 De la spécification abstraite d'un agrégat à sa mise en œuvre polymorphe

Dissociation des spécifications abstraite et opérationnelle(s)

Le principe de l'abstraction de données a été décrit dans plusieurs ouvrages, parmi lesquels [2] et [28]. L'idée fondamentale est de bâtir une hiérarchie de niveaux d'abstraction. Les programmes d'application sont organisés de manière à opérer sur des « données abstraites », dont la représentation concrète est définie indépendamment des programmes qui les utilisent. Cette notion de « donnée abstraite » n'est pas sans analogie avec la notion de type abstrait de données présentée dans [92].

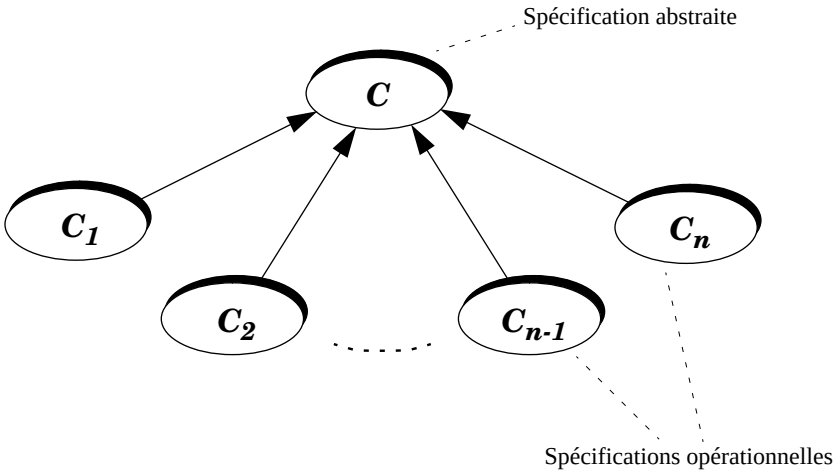


FIG. 1.2 - *Dissociation des spécifications abstraite et opérationnelles grâce à l'héritage*

Pour développer des bibliothèques d'agrégats polymorphes — qu'il s'agisse d'agrégats distribués ou non — à l'aide d'EPEE, on s'appuie sur le principe de l'abstraction de données. Les langages à objets conviennent particulièrement bien à ce type de développement. On dispose de tous les mécanismes nécessaires pour dissocier la spécification abstraite d'une structure de données des détails relatifs à sa mise en œuvre. La spécification abstraite d'un agrégat peut ainsi être encapsulée dans une classe « abstraite » C dont l'interface détermine avec précision la vision qu'aura le programmeur d'application de cet agrégat. On peut ensuite développer à volonté

des classes descendantes (c'est-à-dire des classes héritant de la classe abstraite) C_1, C_2, ... C_n encapsulant chacune une mise en œuvre possible pour le type d'agrégat considéré (voir figure 1.2). Par mise en œuvre, on entend ici aussi bien la description d'un format de représentation en mémoire sur une machine mono-processeur, que la description d'une politique de distribution de l'agrégat sur une architecture parallèle à mémoire distribuée.

Maintien d'une interface homogène

En développant une hiérarchie de classes décrivant un nouveau type d'agrégat, on doit s'attacher à déclarer au plus haut niveau de cette hiérarchie — c'est-à-dire dans la classe encapsulant la spécification abstraite de l'agrégat — toutes les routines devant permettre à un utilisateur de manipuler cet agrégat. En règle générale, on s'interdira également de faire apparaître dans des classes descendantes d'une classe d'agrégats abstraite C de nouvelles routines accessibles à l'utilisateur, ou d'altérer dans ces classes la *signature* des routines déclarées dans C.

Ces contraintes permettent de garantir le maintien d'une interface homogène commune à tous les agrégats partageant une même spécification abstraite : du point de vue de l'utilisateur, tous ces agrégats peuvent être manipulés de manière identique à l'aide des seules routines appartenant à l'interface de la classe abstraite.

1.3.3.3 Vers des agrégats polymorphes distribués

Lorsque la spécification abstraite d'un certain type d'agrégat a été encapsulée dans une classe abstraite, la tâche du concepteur désireux de proposer une mise en œuvre distribuée pour cet agrégat peut être décomposée en deux temps. Dans un premier temps, il lui faut assurer la distribution effective de l'agrégat considéré, tout en garantissant la transparence de cette distribution vis à vis de l'utilisateur. Sa seconde tâche s'inscrit dans le cadre de la recherche de performances. Elle consiste à appliquer aux routines associées aux agrégats distribués certaines techniques d'optimisation et de parallélisation afin de tirer parti de la distribution des données.

Mise en œuvre de la distribution

L'approche expérimentée actuellement dans EPEE étant celle de la parallélisation par distribution des données, chaque nœud de l'APMD cible ne doit posséder qu'une partie d'un agrégat distribué (la ou les *partitions* dont il est propriétaire).

Le concepteur de classes d'agrégats distribués doit donc choisir un ou plusieurs schémas de distribution pour le type d'agrégat considéré. Ayant fait ce choix, il lui faut trouver comment représenter les partitions locales en mémoire. Il lui faut enfin mettre en œuvre des mécanismes assurant, du point de vue de l'utilisateur,

un accès transparent aux données distantes, tout en préservant la sémantique des accès locaux. Une approche communément adoptée consiste à appliquer la règle dite des « écritures locales » (*owner write rule*[5]). Cette règle stipule que seul le nœud propriétaire d'une partition d'un agrégat distribué est autorisé à en modifier le contenu. Elle a été introduite par Calahan et Kennedy dans [26] (sans d'ailleurs qu'ils lui donnent ce nom) et reprise depuis lors dans tous les travaux visant au développement de compilateurs-paralléliseurs pour langages de type HPF.

Le modèle de programmation et d'exécution offert à l'utilisateur d'agrégats distribués étant le modèle SPMD, tout accès en écriture à une donnée V d'un agrégat distribué peut donc être exprimé de la manière suivante :

si je possède (V) **alors**
    modifier localement la valeur de V
fsi

Ce mécanisme très simple assure le respect de la règle des écritures locales. Il est parfois référencé dans la communauté du calcul parallèle par distribution des données sous l'appellation de mécanisme *Exec* [10]. Il conditionne en effet l'*exécution* d'une opération d'écriture en fonction de la propriété des données impliquées dans cette opération[6].

De manière similaire, le mécanisme baptisé *Refresh* assure la transparence des mouvements de données entre les nœuds participant à une exécution répartie : une donnée est « rafraîchie » sur l'ensemble des nœuds avant toute opération de lecture. Une implantation possible du *Refresh* est la suivante :

si je possède (V) **alors**
    diffuser la valeur de V et retourner cette valeur
sinon
    attendre du propriétaire de V la valeur de V
    et retourner cette valeur
fsi

Il a été démontré formellement dans [13, 25] que, partant d'un programme séquentiel, on peut obtenir un programme parallèle sémantiquement équivalent en

5. Encore désignée dans certains documents sous l'appellation de *local update rule* [112, 83].

6. En fait, le mécanisme *Exec* appliqué dans un contexte de parallélisation semi-automatique de code de type HPF peut conditionner, non seulement l'opération d'écriture du résultat d'un calcul, mais aussi le calcul proprement dit. La règle appliquée n'est plus alors la simple règle des *écritures locales* (*owner write rule*), mais la règle des *calculs locaux* (*owner compute rule*).

appliquant de manière systématique les mécanismes de l'*Exec* et du *Refresh* au niveau de tous les accès aux données réalisés dans le programme séquentiel.

On montrera dans le chapitre 3 comment les mécanismes *Exec* et *Refresh* peuvent être encapsulés dans le corps de certaines routines pour assurer la transparence de la distribution d'un agrégat. On verra en outre qu'il n'est pas nécessaire de mettre en œuvre les mécanismes *Exec* et *Refresh* dans *toutes* les routines associées à un type d'agrégat pour réaliser la distribution de cet agrégat, mais que seulement un très petit nombre de routines sont réellement dépendantes de la distribution de l'agrégat et doivent être mises en œuvre en conséquence.

Recherche de performances

Après avoir assuré la distribution d'un agrégat tout en préservant son interface séquentielle, le concepteur peut commencer à procéder de manière incrémentale à l'optimisation des routines associées à cet agrégat. Les techniques d'optimisation intervenant à ce stade sont les mêmes que celles que l'on tente de faire appliquer automatiquement par les compilateurs-paralléliseurs pour langages de type HPF. Elles visent essentiellement à la répartition des calculs (de manière à ce que chaque nœud prenne en charge une partie des calculs devant être réalisés au cours de l'exécution de la routine considérée), et à l'optimisation des échanges de données.

On présentera au chapitre 4 les diverses techniques permettant d'améliorer les performances des agrégats distribués mis en œuvre avec EPEE, en les illustrant dans le cas des matrices distribuées. On verra que la recherche de performances amène au développement de routines optimisées capables d'exploiter au mieux certaines caractéristiques des agrégats distribués impliqués dans les calculs. On montrera comment la sélection dynamique transparente des routines peut être obtenue, afin que l'utilisateur n'ait pas à choisir explicitement la routine la plus adaptée pour réaliser un calcul donné.

Chapitre 2

La bibliothèque de démonstration Paladin

La méthode de conception de bibliothèques parallèles proposée dans le projet EPEE se veut une approche générale. Elle n'est aucunement limitée à la seule conception de bibliothèques manipulant des structures de données régulières, ni à la parallélisation d'algorithmes réguliers, comme en attestent d'ailleurs les résultats de travaux récents portant sur la parallélisation d'un serveur de routage SMDS pour réseau ATM [57], et sur la distribution de graphes d'accessibilité dans l'outil OPEN/CÆSAR [1]. Toutefois, avant d'aborder l'étude des problèmes irréguliers, il nous a paru intéressant de chercher à mieux cerner les possibilités offertes par les langages à objets séquentiels dans le domaine du calcul parallèle, en considérant un domaine d'application régulier et déjà bien connu car largement étudié : celui du calcul d'algèbre linéaire. Nous avons donc entrepris de développer une bibliothèque de démonstration baptisée Paladin [61], permettant d'effectuer des calculs d'algèbre linéaire sur architectures parallèles à mémoire distribuée.

Le domaine de l'algèbre linéaire présente certaines caractéristiques qui en font un sujet d'expérimentation intéressant pour le projet EPEE :

- Les vecteurs et matrices sont des « objets » qu'il est relativement facile de caractériser sous la forme de types de données abstraits, et dont les propriétés sont bien connues. Il est donc relativement facile de les décrire sous forme de classes dans un langage à objets.

- Ces objets répondent bien à notre définition des agrégats polymorphes, dans la mesure où ils « agrègent » un grand nombre de données élémentaires homogènes, peuvent être représentés en mémoire de multiples façons, et peuvent

avoir à changer de format de représentation interne dynamiquement en fonction des besoins des calculs.

- Les algorithmes d'algèbre linéaire sont variés et nombreux. On peut donc encapsuler dans les classes décrivant les agrégats vecteurs et matrices un grand nombre de routines permettant de procéder à des calculs avec ces agrégats.

- Ces algorithmes se prêtent généralement bien (tout au moins lorsque les objets manipulés sont des matrices et vecteurs denses) à la parallélisation par distribution des données. Dans la première phase de développement de la bibliothèque Paladin, nous avons donc choisi de nous intéresser en priorité à ce type de parallélisation, la deuxième phase devant être consacrée à l'intégration dans Paladin de mécanismes permettant la parallélisation des calculs par distribution du contrôle, grâce à l'interfaçage de la bibliothèque avec un système de mémoire virtuelle partagée. Quelques expériences préliminaires ont d'ailleurs déjà été réalisées dans cette voie [71, 81], et la thèse récemment débutée par Jean-Lin Pacherie dans le cadre du projet Pampa porte sur ce thème.

Travaux connexes

Aux États-Unis, des travaux sont en cours qui visent au développement de la bibliothèque ScaLAPACK++ [44], homologue dans le monde C++ de la bibliothèque Fortran ScaLAPACK évoquée au paragraphe 1.3.1. Bien que ScaLAPACK++ soit souvent présentée dans la littérature comme étant une bibliothèque « orientée objets », elle est pourtant loin de mettre pleinement à profit toute la puissance des mécanismes de programmation par objets. ScaLAPACK++ est en fait bâtie au dessus de bibliothèques et de modules pré-existants tels que LAPACK++ (homologue en C++ de la bibliothèque séquentielle LAPACK), PB-BLAS [37], BLACS [115], BLAS [88], etc. Elle est en outre conçue dans la même optique que ScaLAPACK, c'est-à-dire la recherche de performances maximales, et souffre des mêmes lacunes qui ont été évoquées au paragraphe 1.3.1 à propos de ScaLAPACK : redondance du code due à l'absence de généricité[1] et manque d'ouverture vis à vis de l'utilisateur.

En Allemagne, J. Wolff von Gudenberg a récemment entrepris le développement en C++ d'une bibliothèque parallèle d'algèbre linéaire [118]. Dans cette bibliothèque, la mise en œuvre des matrices et vecteurs s'appuie sur la notion de *collec-*

1. En utilisant le mécanisme des *templates* de C++, il serait possible de faire de ScaLAPACK++ une bibliothèque réellement générique, mais il faudrait toutefois que soient réécrites en C++ et de manière générique les modules BLAS, PB-BLAS, etc. sur lesquels s'appuie la mise en œuvre de ScaLAPACK++.

tion, et l'accent est mis sur l'implantation d'algorithmes s'*auto-vérifiant* (c'est-à-dire fournissant non seulement le résultat d'un calcul mais aussi la précision de ce résultat).

2.1 Vue d'ensemble de la bibliothèque

La méthode de développement des agrégats polymorphes que nous préconisons dans le cadre du projet EPEE s'appuie sur la dissociation des spécifications abstraite et opérationnelles d'un agrégat. La bibliothèque Paladin s'articule donc autour de deux classes Eiffel décrivant les entités élémentaires de l'algèbre linéaire : les vecteurs et les matrices.

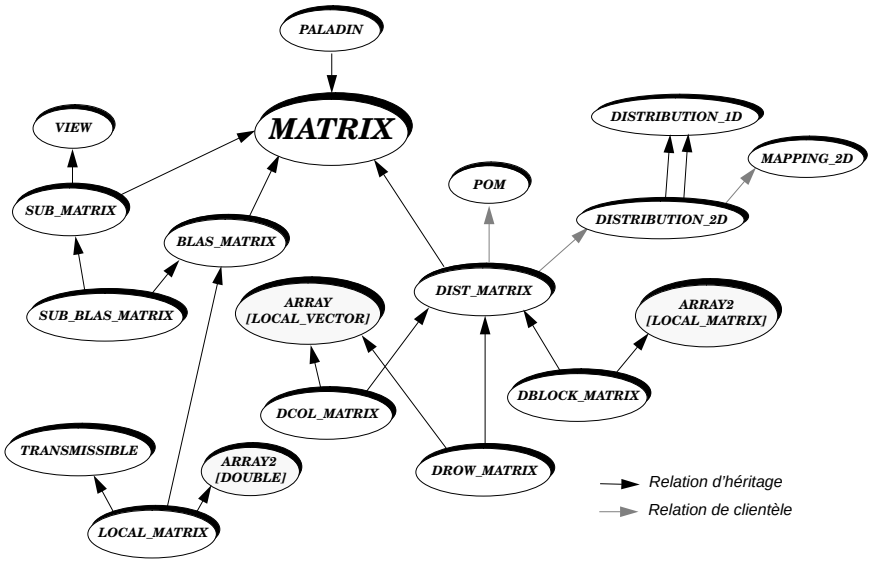


FIG. 2.1 - *Hiérarchie des classes Eiffel décrivant les matrices (vue partielle)*

On a reproduit dans la figure 2.1 une vue d'ensemble de la hiérarchie des classes Eiffel décrivant les agrégats matrices dans la bibliothèque Paladin. La plupart des classes apparaissant dans cette figure seront détaillées au fil des pages dans ce document. À chaque fois, un sous-ensemble de la hiérarchie de la figure 2.1 sera reproduit pour permettre au lecteur de mieux situer comment la classe considérée s'inscrit dans la hiérarchie globale de Paladin.

Dans la figure 2.1, on voit apparaître la classe « abstraite » MATRIX. Cette classe est — avec son homologue la classe VECTOR — celle qui offre le plus haut niveau d'abstraction dans la bibliothèque : aucun détail n'y est donné quant à la manière de représenter des objets matrices en mémoire. Tous les détails relatifs à la

représentation en mémoire des agrégats matrices ont volontairement été ignorés à ce niveau, afin que plusieurs mises en œuvre alternatives puissent être décrites dans les classes descendantes de la classe abstraite MATRIX.

La bibliothèque comprend également une hiérarchie assez semblable à celle reproduite dans la figure 2.1, mais décrivant les agrégats vecteurs. Dans les paragraphes et chapitres qui suivent, on se focalisera essentiellement sur la mise en œuvre des agrégats matrices, celle des agrégats vecteurs n'étant évoquée qu'épisodiquement.

2.2 Spécification abstraite des agrégats matrices

La classe MATRIX a pour fonction essentielle de fixer l'*interface* des agrégats matrices, c'est-à-dire que nous y avons *déclaré* toutes les primitives (attributs et routines) avec lesquelles un utilisateur pourra *manipuler* ces objets.

2.2.1 Interface

On a reproduit dans l'exemple 2.1 une partie de l'interface de la classe MATRIX. Dans cette interface apparaissent les spécifications des primitives (attributs et routines) permettant à un utilisateur de manipuler un objet matrice. Chaque primitive est caractérisée par son nom, sa signature, et éventuellement par des préconditions et postconditions.

Note sur les assertions

Nous avons introduit un grand nombre d'assertions dans les classes constituant la bibliothèque Paladin (voir par exemple les préconditions et postconditions associées aux routines *put* et *item* dans l'exemple 2.1). Cependant, ces assertions ne seront pas forcément toutes apparentes dans les exemples de code reproduits dans ce document. La plupart du temps n'apparaitront dans ces exemples que les assertions susceptibles d'informer le lecteur et d'aider à sa compréhension.

2.2.2 Classification des routines

Il existe plusieurs manières de classifier les opérations associées à un type abstrait de données. Liskov et Guttag proposent par exemple de distinguer entre les *constructeurs primitifs*, les (autres) *constructeurs*, les *mutateurs* et les *observateurs* [92]. Nous n'avons pas besoin de distinguer autant de « familles » d'opérations dans les classes

Exemple 2.1

```
deferred class interface MATRIX

feature -- Attributes
   nrow: INTEGER      -- Number of rows in Current
   ncolumn: INTEGER   -- Number of columns in Current

feature -- Basic Accessors
   item (i, j: INTEGER): DOUBLE
         -- Return current value of item(i, j)
      require
         valid_i: (i > 0) and (i <= nrow)
         valid_j: (j > 0) and (j <= ncolumn)
   put (v: like item; i, j: INTEGER)
         -- Put value v into item(i, j)
      require
         valid_i: (i > 0) and (i <= nrow)
         valid_j: (j > 0) and (j <= ncolumn)
      ensure
         item (i, j) = v

feature -- High-Level Accessors
   row (i: INTEGER): VECTOR
      -- Provide a view on i-th row
   column (j: INTEGER): VECTOR
      -- Provide a view on j-th column
   diagonal (k: INTEGER): VECTOR
      -- Provide a view on k-th diagonal
   submatrix (i, j, k, l: INTEGER): MATRIX
      -- Provide a view on submatrix (i:j,k:l)

feature -- Operators
   trace: like item
      -- Trace of current matrix
   random (min, max: like item)
      -- Random initialization
   add (B: MATRIX)
      -- Matrix addition (Current <- Current + B)
   mult (A, B: MATRIX)
      -- Matrix multiply (Current <- A * B)
   mgs (R: MATRIX)
      -- Q.R Decomposition of Current (Q overwrites Current)
   ...
end -- MATRIX
```

Point de langage 2.1

Typage par ancrage. Le mécanisme de typage par ancrage est une facilité syntaxique permettant de spéficier que, dans une classe donnée et dans toutes ses descendantes, une ou plusieurs entités sont du même type qu'une entité de référence jouant le rôle d'« ancre » pour le typage. Ainsi, dans la classe MATRIX (voir l'exemple 2.1), le type du résultat de la fonction *item* sert d'ancre pour la plupart des autres routines de la classe.

de Paladin. Nous nous contentons de grouper les routines permettant de manipuler un agrégat en deux familles principales : la famille des accesseurs et celle des opérateurs.

Les accesseurs

Les accesseurs sont des routines permettant d'accéder au contenu d'un agrégat en lecture ou en écriture. Les routines *put* et *item* déclarées dans la classe MATRIX sont les *accesseurs de base* de cette classe : elles permettent d'accéder directement à l'un des éléments scalaires d'un objet matrice.

On pourra noter dans l'exemple 2.1 que le type de l'argument v de la routine *put* est spécifié par référence au type du résultat de *item*. Il s'agit là d'une application du mécanisme de typage par ancrage propre au langage Eiffel. Ce mécanisme est décrit dans le point de langage 2.1.

Outre les accesseurs de base *put* et *item*, nous avons également introduit dans la classe MATRIX des accesseurs de plus haut niveau permettant de manipuler une colonne, une ligne ou une diagonale de la matrice courante comme étant un objet de type vecteur (c'est-à-dire un objet dont le type est conforme au type VECTOR), et une section rectangulaire de la matrice courante comme étant un objet de type matrice. Ces accesseurs de haut niveau fournissent à peu près la même abstraction que les raccourcis syntaxiques qui sont fréquemment utilisés par les auteurs d'ouvrages traitant d'algèbre linéaire [55]. Ainsi, si A désigne une matrice $n \times m$, l'expression *A.submatrix(i,j,k,l)* équivaut à la notation $A(i : j, k : l)$. De même, *A.row(i)* et *A.column(j)* sont équivalentes à $A(i, :)$ et $A(:, j)$ respectivement.

Une caractéristique majeure des accesseurs de haut niveau est que leur utilisation n'implique pas de copie systématique des données. Ces accesseurs fournissent simplement une « vue » sur une section particulière de la matrice spécifiée. Modifier cette vue équivaut à modifier directement la section correspondante de la matrice englobante. Pour illustrer ce mécanisme, supposons que A désigne une matrice 5×5

nouvellement créée, dont tous les éléments sont initialement nuls[2]. La figure 2.2 montre comment l'accesseur *submatrix* peut être utilisé pour remplir de manière aléatoire une section rectangulaire de la matrice A. Dans cet exemple, la section $A(2:4,2:5)$ est remplie aléatoirement avec des valeurs prises dans l'intervalle $[1,0..9,0]$.

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \xrightarrow{\text{A.submatrix(2,4,2,5).random(1.0,9.0)}} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 1.2 & 9.8 & 6.2 \\ 0 & 2.1 & 7.0 & 2.4 & 8.4 \\ 0 & 7.8 & 3.9 & 5.7 & 1.8 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

FIG. 2.2 - *Emploi de l'accesseur* submatrix *pour modifier une section rectangulaire de la matrice A*

- **Note**. Le lecteur désireux de connaître les détails de mise en œuvre du mécanisme des vues dans Paladin pourra se reporter à l'annexe A. Toutefois, il n'est pas indispensable de comprendre *comment* les vues fonctionnent pour pouvoir poursuivre la lecture de ce document.

Les accesseurs de haut niveau – et le mécanisme sous-jacent des vues — apportent un confort indéniable au programmeur, que ce soit pour incorporer de nouveaux opérateurs dans les classes MATRIX et VECTOR, ou pour manipuler des matrices et vecteurs au niveau d'un programme d'application.

La variété des accesseurs permet d'exprimer n'importe quel algorithme d'algèbre linéaire au niveau d'abstraction désiré. L'incorporation de nouveaux algorithmes dans la bibliothèque s'en trouve grandement facilitée : la plupart des algorithmes d'algèbre linéaire peuvent être « traduits » très aisément en Eiffel et intégrés aussitôt dans les classes de Paladin.

Outre qu'elle permet de privilégier le confort du programmeur en adoptant un style d'écriture des algorithmes particulièrement lisible, la variété des accesseurs mis en œuvre dans Paladin permet également de choisir les accesseurs en fonction des caractéristiques de la représentation interne des opérandes manipulées dans un algorithme (et notamment de leur distribution éventuelle). Cet aspect est abordé plus en détails dans le chapitre 4.

2. À la création d'un objet Eiffel, des valeurs par défaut sont affectées à tous les champs de cet objet.

Point de langage 2.2

Invocation d'une primitive. L'invocation (ou l'*appel*) est l'application d'une primitive donnée à un objet donné, avec éventuellement des arguments. Ceci s'exprime avec la notation pointée. L'invocation *M.trace* demande donc à ce que la fonction *trace* soit appliquée à (ou *invoquée sur*) l'objet référencé par *M*, de même que l'invocation *A.add(B)* demande à ce que la procédure *add* soit invoquée sur l'objet référencé par *A* avec le paramètre *B*. Lorsqu'aucune *cible* (c'est-à-dire l'entité désignée à gauche du point) n'est spécifiée dans l'invocation, la cible implicite est alors l'objet courant (qui peut être désigné par le mot-clé **Current** en Eiffel).

Les opérateurs

On qualifie d'opérateurs toutes les routines permettant à un utilisateur de réaliser des calculs impliquant un agrégat pris dans son ensemble, ainsi que d'autres paramètres éventuels. Pour la plupart des opérateurs déclarés dans la classe MATRIX, il existe un homologue mathématique dans le domaine de l'algèbre linéaire. Ainsi, l'opérateur *trace* est une fonction qui calcule et retourne la somme des éléments diagonaux d'une matrice carrée M lorsqu'on invoque sur cette matrice la routine *trace* dans une expression pointée de la forme $v := M.trace$. De même, l'opérateur *add* permet d'additionner une matrice B à une matrice de même taille A à l'aide de l'expression pointée *A.add(B)*. L'opérateur *mult* permet quant à lui de calculer un produit de matrices de la forme $C = A \times B$ en utilisant l'expression *C.mult(A,B)*. Le lecteur non familiarisé avec le principe de la notation pointée pourra se référer utilement au point de langage 2.2.

On n'a reproduit dans l'exemple 2.1 que les signatures de quelques uns des opérateurs de la classe MATRIX. Nous avons en fait introduit dans cette classe un grand nombre d'opérateurs — une quarantaine dans la version actuelle de Paladin — qui permettent à un utilisateur d'effectuer la plupart des opérations d'algèbre linéaire traditionnelles, telles que les opérations de type scalaire-matrice, matrice-vecteur et matrice-matrice (somme, différence, produit, transposée, etc.), mais aussi des opérations plus complexes telles que les factorisations de type LU, LDL^T, et QR, la résolution de systèmes triangulaires, etc.

La classe VECTOR, qui est le pendant de la classe MATRIX pour les entités de type vecteur, renferme de même un grand nombre d'opérateurs pour effectuer des opérations de type scalaire-vecteur et vecteur-vecteur.

Point de langage 2.3

Routines et classes différées. Une routine « différée » dans une classe Eiffel équivaut à une fonction « virtuelle pure » dans une classe C++ : il s'agit d'une routine pour laquelle aucune définition (c'est-à-dire aucune mise en œuvre) n'est fournie, et qui doit donc impérativement être définie dans une classe descendante. Une classe contenant au moins une routine différée est elle-même une classe différée : elle ne peut être instanciée, mais on peut l'utiliser pour développer des classes descendantes non différées (classes *concrètes* ou *effectives*).

2.2.3 Abstraction pure et abstraction partielle

La classe MATRIX étant une classe *abstraite*, aucun détail ne peut y être donné quant à la manière de représenter les matrices en mémoire. Nous avons volontairement ignoré ce genre de détail de mise en œuvre au niveau de cette classe, afin qu'aucune contrainte de mise œuvre ne soit imposée sur les classes descendantes devant être construites.

Ceci ne signifie cependant pas qu'aucun accesseur ni opérateur ne peut être défini dans la classe MATRIX. En fait, dans cette classe seuls les accesseurs de base *put* et *item* doivent impérativement être maintenus différés (voir éventuellement le point de langage 2.3) au niveau de cette classe, car eux seuls ont une mise en œuvre dépendant étroitement du format choisi pour représenter un objet matrice en mémoire. Nous avons déclaré les accesseurs *put* et *item* au niveau de la classe MATRIX, mais nous ne les y avons pas définis.

À la différence de *put* et *item*, les accesseurs de plus haut niveau et les opérateurs sont des primitives dont la définition peut fort bien être faite sur la base d'appels aux accesseurs de base, voire d'appels à d'autres accesseurs ou opérateurs. La mise en œuvre d'un opérateur, par exemple, n'est donc pas directement dépendante du format de représentation choisi, celui-ci étant masqué par les deux accesseurs de base.

À titre d'exemple, on montre dans les paragraphes suivants comment nous avons défini les opérateurs *trace*, *mgs* et *mult* dans la classe MATRIX. Dans ces paragraphes, la mise en œuvre de ces opérateurs est détaillée à l'intention des lecteurs peu familiarisés avec les langages à objets en général, et avec le langage Eiffel en particulier. On utilisera les mêmes exemples dans le chapitre 4 pour illustrer la parallélisation de la bibliothèque Paladin.

Exemple 1 : définition séquentielle de l'opérateur *trace*

La trace d'une matrice carrée A est la somme de ses éléments diagonaux, définie formellement par : $trace(A) = \sum_{i=1}^{n} a_{ii}$. Dans la classe MATRIX, nous avons doté la fonction *trace* de l'algorithme séquentiel reproduit dans l'exemple 2.2 :

Exemple 2.2

```
trace: like item is
        -- Sums up all diagonal items
    require
        is_square: (nrow = ncolumn);
    local
        i: INTEGER;
    do
        from i := 1 until i > nrow loop
            Result := Result + item (i, i);
            i := i + 1
        end; -- loop
    end;
```

La trace d'une matrice ne peut être calculée que si cette matrice est carrée. Cette contrainte est exprimée par la précondition *is_square* associée à la routine *trace* dans l'exemple précédent (ligne 4).

L'algorithme de la routine *trace* s'appuie sur une itération : on énumère toutes les valeurs de i comprises entre 1 et *nrow* (nombre de lignes dans la matrice courante) et on calcule la somme des éléments diagonaux dans cette matrice. On accède à la valeur d'un élément (i, i) donné en invoquant l'accesseur *item* sur la matrice courante.

Exemple 2 : définition séquentielle de l'opérateur *mgs*

L'algorithme dit « de Gram-Schmidt modifié » permet de décomposer une matrice $A_{m \times n}$ de rang n en un produit de deux matrices $Q.R$ tel que les colonnes de $Q_{m \times n}$ soient orthonormales et $R_{m \times n}$ soit triangulaire supérieure. En pratique, cet algorithme est souvent mis en œuvre de telle manière que la matrice Q résultant de la décomposition de la matrice courante A recouvre la matrice A. On évite ainsi d'avoir à créer une nouvelle matrice de taille $m \times n$ pour stocker Q, mais on perd en revanche l'information contenue initialement dans A.

L'algorithme de factorisation de Gram-Schmidt (avec Q recouvrant A) est ex-

primé en pseudo-code dans [55] comme illustré ci-dessous. On a numéroté les phases de calcul principales afin de pouvoir faire le lien avec le code Eiffel décrit plus loin :

pour k **depuis** 1 **jqa** n **faire**
  $R(k,k) := \|A(:,k)\|_2$ (1)
  $A(:,k) := A(:,k)/R(k,k)$ (2)
  pour j **depuis** $k+1$ **jqa** n **faire**
    $R(k,j) := A(:,k)^T A(:,j)$ (3)
    $A(:,j) := A(:,j) - A(:,k).R(k,j)$ (4)
  fpour
fpour

Nous avons traduit cet algorithme séquentiel en Eiffel et avons intégré le code résultant de cette traduction dans le corps de l'opérateur *mgs*[3] de la classe MATRIX. Le code de cet opérateur est reproduit dans l'exemple 2.3.

L'opérateur *mgs* décompose la matrice sur laquelle il est invoqué en un produit de matrices $Q.R$, Q recouvrant la matrice courante et R recouvrant la matrice passée en paramètre à l'opérateur. Les opérations réalisées dans le corps de cet opérateur sont détaillées ci-dessous :

- Ligne *(1)* : calcul de $R(k,k) \leftarrow \|A(:,k)\|_2$

  - On invoque l'accesseur *column* pour obtenir une vue sur le $k^{ème}$ vecteur colonne de la matrice courante.

    column(k)

  - On invoque sur la vue retournée par *column(k)* la fonction *nrm2* afin d'obtenir la norme de la colonne k. L'opérateur fonction *nrm2* est défini dans la classe VECTOR. Il retourne la valeur de la norme du vecteur sur lequel il est invoqué.

    column(k).nrm2

  - Le résultat retourné par *nrm2* est affecté à l'élément d'indices (k,k) de la matrice R en invoquant sur R l'accesseur *put*.

    R.put (column(k).nrm2, k, k)

- Ligne *(2)* : calcul de $A(:,k) \leftarrow A(:,k)/R(k,k)$

3. mgs : *modified Gram-Schmidt.*

Exemple 2.3

```
deferred class MATRIX
   ...
feature -- Operators
   mgs (R: MATRIX) is
         -- Modified Gram-Schmidt (Q.R decomposition)                    5
         -- Q.R <- Current (Q overwrites Current)
      require
         rank_ok: (Current.rank = ncolumn);
         size_ok: (nrow = R.nrow) and (nrow = R.ncolumn);
      local                                                              10
         k, j: INTEGER;
      do
         from k:= 1 until k > ncolumn loop
            R.put (column(k).nrm2, k, k);       -- (1)
            column(k).scal (1.0 / R.item(k, k));       -- (2)             15
            from j:= k+1 until j > ncolumn loop
               R.put (column(k).dot(column(j)), k, j);       -- (3)
               column(j).axpy (- R.item(k, j), column(k));       -- (4)
               j:= j + 1;
            end; -- loop                                                 20
            k:= k + 1;
         end; -- loop
      end; -- mgs
   ...
end -- MATRIX                                                            25
```

On doit ici diviser chacun des éléments de la colonne k de la matrice courante par $R(k,k)$.

- On calcule donc l'inverse de $R(k,k)$...

$$\textit{1.0 / R.item(k, k)}$$

- et on passe le résultat en paramètre à l'opérateur *scal* invoqué sur une vue du $k^{ème}$ vecteur colonne de la matrice. L'opérateur *scal* est défini dans la classe VECTOR. Il multiplie tous les éléments du vecteur sur lequel il est invoqué par une valeur scalaire passée en paramètre.

$$\textit{column(k).scal (1.0 / R.item(k, k))}$$

- Ligne *(3)* : calcul de $R(k,j) \leftarrow A(:,k).A(:,j)$

On doit évaluer le produit scalaire des vecteurs colonnes k et j de la matrice courante et placer le résultat dans $R(k,j)$.

- On invoque donc l'opérateur fonction *dot* afin de calculer le produit scalaire des deux colonnes considérées. L'opérateur fonction *dot*, défini dans la classe VECTOR, calcule le produit scalaire du vecteur sur lequel il est invoqué et d'un second vecteur passé en paramètre.

$$\textit{column(k).dot (column(j))}$$

- Le résultat est affecté à l'élément d'indices (k,j) de la matrice R grâce à une invocation de l'accesseur *put* sur cette matrice.

$$\textit{R.put (column(k).dot(column(j)), k, j)}$$

- Ligne *(4)* : calcul de $A(:,j) \leftarrow A(:,j) - A(:,k).R(k,j)$

On doit réaliser ici une opération de type *saxpy* (*scalar alpha x plus y*). On invoque sur une vue de la colonne j de la matrice courante l'opérateur *axpy* défini dans la classe VECTOR, en lui passant en paramètres l'opposé de $R(k,j)$ et une vue sur la colonne k de la matrice courante. L'opérateur *axpy*, invoqué dans une expression de la forme *y.axpy(a,x)*, réalise un produit scalaire-vecteur de la forme $y \leftarrow a.x + y$.

$$\textit{column(j).axpy (- R.item(k, j), column(k))}$$

Exemple 3 : définition séquentielle de l'opérateur *mult*

Il va de soi que l'algorithme du produit de matrices peut être exprimé de multiples façons. Nous avons choisi de définir l'opérateur *mult* dans la classe MATRIX comme illustré dans l'exemple 2.4.

Exemple 2.4

```
mult (A, B : MATRIX) is
        -- Matrix multiply (Current <- A * B)
    require
        size_ok : (nrow = A.nrow) and (ncolumn = B.ncolumn)
                  and (A.ncolumn = B.nrow);
    do
        from i := 1 until i > C.nrow loop
            from j := 1 until j > C.ncolumn loop
                put (A.row (i).dot (B.column (j)), i, j);
                j := j + 1;
            end; -- loop
            i := i + 1;
        end; -- loop
    end; -- mult
```

L'opérateur permet de calculer le produit de trois matrices $C \leftarrow A \times B$ à l'aide de l'expression pointée $C.mult(A, B)$. On notera dans l'exemple 2.4 la précondition qui précise les conditions d'utilisation de cet opérateur.

Pour définir l'opérateur *mult*, nous avons choisi d'utiliser les accesseurs *row* et *column* de la classe MATRIX et l'opérateur *dot* de la classe VECTOR. Ainsi, l'algorithme encapsulé dans l'opérateur *mult* consiste simplement en une « traduction » en Eiffel de l'algorithme exprimé ci-dessous en pseudo-code :

pour i **depuis** 1 **jqa** m **faire**
  pour j **depuis** 1 **jqa** n **faire**
    $C(i,j) := A(i,:)^T.B(:,j)$
  fpour
fpour

Nous aurions fort bien pu décider de ne pas utiliser les accesseurs vectoriels offerts par la classe MATRIX et d'exprimer l'algorithme de calcul du produit de matrices à l'aide de trois boucles imbriquées et de simples appels aux opérateurs *put* et *item*. Nous aurions également pu choisir d'accéder plutôt aux colonnes de la matrice A et aux lignes de la matrice B. En fait, le type d'algorithme finalement

encapsulé dans l'opérateur *mult* de la classe MARTIX importe relativement peu. Seul compte le fait que cet algorithme réalise bien le calcul désiré. Si cet algorithme devait s'avérer peu performant à l'usage, il serait toujours possible d'en modifier la mise en œuvre dans l'opérateur *mult* sans perturber l'ensemble des classes constituant la bibliothèque Paladin ni les classes « clientes » manipulant des matrices dans le cadre d'une application.

Il faut garder à l'esprit que l'algorithme encapsulé dans l'opérateur *mult* doit être considéré comme un algorithme séquentiel par défaut, capable de calculer un produit de matrices $C = A \times B$ quelles que soient les caractéristiques de représentation interne des trois objets A, B et C impliqués dans le calcul. Cet algorithme étant purement séquentiel, il n'est absolument pas adapté pour calculer efficacement le produit de matrices distribuées. On montrera au chapitre 4 comment pour chaque opérateur des classes MATRIX et VECTOR on peut être amené à proposer plusieurs mises en œuvres alternatives, chaque variante étant conçue afin d'exploiter au mieux certaines caractéristiques de la représentation interne — et éventuellement de la distribution — des objets impliqués dans le calcul. On montrera en outre comment la sélection dynamique de l'algorithme le plus approprié pour réaliser le calcul requis peut être assurée de manière transparente pour l'utilisateur.

À l'instar des opérateurs *trace*, *mgs* et *mult* dont les algorithmes par défaut ont été reproduits ci-dessus, tous les opérateurs des classes MATRIX et VECTOR sont dotés d'une mise en œuvre séquentielle par défaut. Les classes MATRIX et VECTOR sont donc qualifiées de classes abstraites, non pas parce qu'on n'y détaille aucune spécification opérationnelle, mais simplement parce qu'on y fait totalement abstraction des détails relatifs à la représentation en mémoire des matrices et vecteurs impliqués dans les calculs.

2.2.4 Extensibilité algorithmique

Il demeure toujours possible d'ajouter dans la classe MATRIX un nouvel opérateur et d'en faire bénéficier l'ensemble de la bibliothèque Paladin. Grâce au mécanisme d'héritage, l'ajout d'un nouvel opérateur au niveau de la classe abstraite MATRIX sera répercuté au niveau de toutes les classes descendant de MATRIX. Si cet opérateur est en outre doté d'une mise en œuvre séquentielle par défaut comme l'ont été tous les opérateurs introduits jusqu'à ce jour dans la classe MATRIX, cette mise en œuvre sera tout aussi valable au niveau des classes décrivant un format de représentation interne pour les matrices.

La classe VECTOR décrivant la spécification abstraite des agrégats vecteurs a été conçue de manière semblable à la classe MATRIX. On peut également y adjoindre à volonté de nouveaux opérateurs.

La bibliothèque Paladin est donc bien totalement extensible — en termes d'extension algorithmique — conformément aux objectifs énoncés au paragraphe 1.3.2.2.

2.2.5 Remarques

À propos de l'indiçage

Dans la classe MATRIX, le domaine d'indices considéré pour désigner un élément scalaire quelconque (i, j) d'une matrice de taille $m \times n$ est tel que $1 \leq i \leq n$ et $1 \leq j \leq n$. Sur ce point, nous avons choisi de nous en tenir aux conventions d'indiçage adoptées dans la plupart des ouvrages traitant d'algèbre linéaire, dans lesquels les domaines d'indices sont généralement bornés en 1 plutôt qu'en 0.

À propos de la généricité

Dans la version actuelle de la bibliothèque Paladin, seule la mise en œuvre des matrices et vecteurs de nombres réels en double précision est considérée. Le langage Eiffel permet pourtant de bâtir des classes génériques, mais en raison d'un défaut dans la mise en œuvre des types numériques de base INTEGER, REAL, et DOUBLE dans l'environnement de compilation ISE utilisé à l'Irisa (version 3.2.3), nous avons dû renoncer *temporairement* à assurer la généricité des classes de Paladin.

En fait, l'une des contributions indirectes du travail de thèse rapporté dans ce document a été de faire évoluer la bibliothèque des classes standard associée au langage Eiffel. En développant la bibliothèque Paladin et en compilant ses classes à l'aide de l'environnement 3.2.3 d'ISE, nous avons décelé un défaut dans le typage de certaines routines déclarées dans la classe NUMERIC[4]. En substance, les signatures de ces routines sont telles qu'il devient pratiquement impossible de développer des classes génériques dont le paramètre générique soit contraint par le type NUMERIC. Avec l'environnement 3.2.3 d'ISE, il est donc notamment impossible de développer des classes décrivant des matrices et vecteurs génériques (notons que ce problème ne se pose pas avec l'environnement TowerEiffel, dans lequel la classe NUMERIC est conçue différemment).

J'ai proposé conjointement avec Mickael Rowley une nouvelle manière d'organiser la hiérarchie des classes numériques dans l'environnement Eiffel. Cette proposition, initialement diffusée sous le titre *Towards a new hierarchy of numeric objects* dans le groupe de messagerie électronique `comp.lang.eiffel`, a ensuite été reprise pour publication dans la revue *Eiffel Outlook*[5].

4. La classe NUMERIC est l'une des classes de la bibliothèque standard d'Eiffel. Elle fait office d'ancêtre commun aux classes INTEGER, REAL, et DOUBLE.

5. Compilation par Robert « Rock » Howard d'une série d'articles extraits du forum électronique `comp.lang.eiffel`, février 1994

Depuis lors, le comité NICE (*Nonprofit International Consortium for Eiffel*), chargé de superviser l'évolution du langage Eiffel et de la bibliothèque standard associée, a publié une proposition d'interface normalisée pour les classes de la bibliothèque standard. Dans cette proposition, baptisée PELKS (*Proposed Eiffel Library Kernel Standard*), l'interface de la classe NUMERIC est dans l'ensemble conforme à nos suggestions, et le problème relatif au typage des routines de la classe NUMERIC a disparu. Par ailleurs, la toute nouvelle version 3.3 de l'environnement Eiffel d'ISE est bâtie sur la bibliothèque PELKS. Il n'y a donc plus aucun obstacle à ce que nous fassions dans un proche avenir de la bibliothèque Paladin une bibliothèque réellement générique.

2.3 Vers des matrices opérationnelles

Dès lors que la spécification abstraite d'un agrégat a été encapsulée dans une classe, il est possible de développer une ou plusieurs classes héritant de la classe abstraite, chaque classe descendante encapsulant une mise en œuvre possible de l'agrégat considéré. Cette mise en œuvre peut simplement consister dans la description d'un format de représentation pour représenter l'agrégat en mémoire sur une machine séquentielle. Il peut aussi s'agir de la description d'un schéma permettant de distribuer l'agrégat sur une APMD.

2.3.1 Une mise en œuvre des matrices locales

Dans la bibliothèque Paladin, les détails relatifs à la représentation en mémoire des matrices sont encapsulés dans des classes descendantes de la classe abstraite MATRIX. Ainsi, la classe LOCAL_MATRIX décrit l'une des mises en œuvre possibles pour les matrices denses locales.

Format de représentation interne

Une matrice de type LOCAL_MATRIX est représentée en mémoire sous la forme d'un simple tableau bi-dimensionnel. La classe LOCAL_MATRIX combine donc simplement la spécification abstraite héritée de la classe MATRIX avec les mécanismes de stockage fournis par la classe ARRAY2, l'une des nombreuses classes standard de la bibliothèque Eiffel (figure 2.3).

La classe LOCAL_MATRIX tient en seulement quelques lignes[6]. On voit là un exemple typique d'utilisation du mécanisme d'héritage multiple : la spécification

6. En fait le code reproduit dans l'exemple 2.5 ne constitue pas la version finale de la classe LOCAL_MATRIX. On sera amené à y ajouter quelques lignes dans les paragraphes et chapitres suivants.

Exemple 2.5

```
class LOCAL_MATRIX inherit
    MATRIX
        redefine nrow, ncolumn end ;
    ARRAY2 [DOUBLE]
        rename height as nrow, width as ncolumn end ;                    5
creation
  make
end -- LOCAL_MATRIX
```

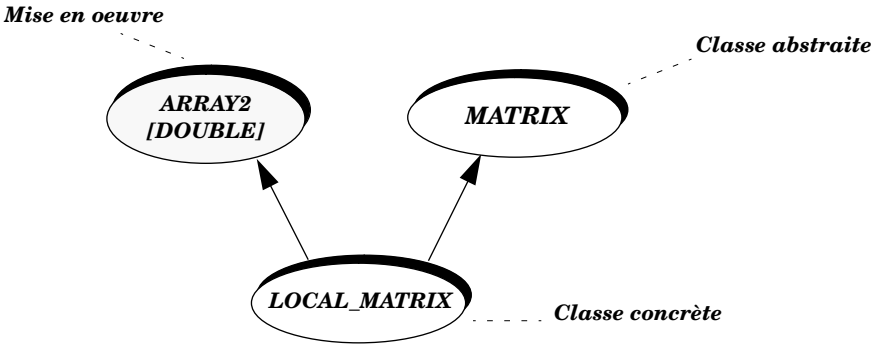


FIG. 2.3 - *Construction de la classe* LOCAL_MATRIX *grâce au mécanisme de l'héritage multiple*

abstraite héritée de la classe MATRIX est combinée avec les facilités de mise en œuvre offertes par une classe standard de la bibliothèque Eiffel. L'effort de développement se limite donc ici à assurer la correspondance entre les noms des routines et attributs hérités des deux classes ancêtres (les attributs *height* et *width* de la classe ARRAY2 sont ici mis en correspondance avec les attributs *nrow* et *ncolumn* de la classe MATRIX).

Avertissement : dans toutes les figures de ce document, on a fait apparaître en grisé les classes Eiffel prises dans la bibliothèque standard du langage (voir par exemple la classe ARRAY2 dans la figure 2.3). Toutes les autres classes, représentées avec un fonc blanc, sont des classes que nous avons développées, soit dans le but de les intégrer à la boîte à outils d'EPEE, soit en tant que composants logiciels de la bibliothèque Paladin.

Exemple d'utilisation

À la différence de la classe abstraite MATRIX, la classe LOCAL_MATRIX est une classe concrète (ou effective), ce qui signifie qu'elle peut être instanciée. On peut donc créer des matrices locales dans un programme d'application, et les manipuler en invoquant les opérateurs hérités de la classe MATRIX, comme illustré dans l'exemple 2.6.

Exemple 2.6

```
local
    A, B, C : MATRIX;
do
   !LOCAL_MATRIX !A.make (10, 10);
   !LOCAL_MATRIX !B.make (10, 10);                                    5
    A.random (−5.0, +5.0); B.random (−5.0, +5.0);
    A.add (B);
    C := A;
end;
```

Dans cet exemple, on crée deux matrices de taille 10×10 et de type LOCAL_MATRIX que l'on affecte aux variables locales A et B. On utilise ici le mécanisme de création avec typage explicite (voir éventuellement le point de langage 2.4, page 60). Ce mécanisme permet d'obtenir qu'un objet de type T nouvellement créé soit référencé par une entité de type U, à condition toutefois que la classe T soit une descendante de la classe U. Ainsi, dans l'exemple 2.6, les *variables locales* A, B, et C sont déclarées comme ayant pour *type statique* le type MATRIX. Ce n'est que lors de la création effective d'objets matrices et de leur affectation à A et de B que ces deux variables prennent le *type dynamique* de l'objet auquel elles sont associées, c'est-à-dire ici le type LOCAL_MATRIX. Il en va de même lors de l'affectation de la variable C, qui prend pour type dynamique le type de l'objet nouvellement référencé, c'est-à-dire ici encore le type LOCAL_MATRIX.

Après avoir créé les deux matrices et les avoir affectées aux variables A et B, on initialise ces matrices de manière aléatoire en invoquant sur chacune d'elles la procédure *random*[7]. On ajoute ensuite la matrice référencée par B à la matrice référencée par A en invoquant sur A l'opérateur *add*[8], et en passant en paramètre à cet opérateur la variable B. On affecte enfin la variable A à la variable C, ce qui a pour conséquence qu'après cette affectation les variables A et C référencent toutes

7. La procédure *random* est définie dans la classe MATRIX.
8. La procédure *add* est également définie dans la classe MATRIX.

Point de langage 2.4

Création d'un objet. Les objets sont créés explicitement à l'aide de la notation *!type !x.rout*, où *type* désigne le type de l'objet devant être créé, *x* désigne l'entité à laquelle l'objet doit être affecté, et *rout* désigne une routine de création devant être invoquée aussitôt après la création de l'objet. On peut parfois omettre de spécifier le type de l'objet désiré. Dans ce cas l'objet créé a pour type le type statique (ou type déclaré) de l'entité *x*.

deux le même objet matrice (il n'y a donc pas duplication de l'objet, mais référence multiple à cet objet).

Vers des matrices locales transmissibles

Les instances de la classe LOCAL_MATRIX jouent un rôle fondamental dans la bibliothèque Paladin. Du point de vue d'un programmeur d'application, un objet de type LOCAL_MATRIX créé au niveau d'un programme SPMD est une matrice locale à tous les nœuds, ce qui revient à dire qu'elle est dupliquée. En tant que concepteur des classes de Paladin, on peut en revanche percevoir une instance de la classe LOCAL_MATRIX, soit comme un objet dupliqué, soit comme un objet local à un nœud particulier.

En développant les classes décrivant les différents types de matrices dans Paladin, nous avons utilisé des objets de type LOCAL_MATRIX en tant que composants logiciels élémentaires pour la mise en œuvre de certaines matrices distribuées (cet aspect est présenté au paragraphe 3.4.1). Pour cette raison, nous avons fait des objets de type LOCAL_MATRIX des objets transmissibles. Pour ce faire, il nous a suffi de modifier la clause d'héritage afin que la classe LOCAL_MATRIX hérite de la classe TRANSMISSIBLE (figure 2.4), et d'implanter dans la classe LOCAL_MATRIX les primitives de communication héritées de la classe TRANSMISSIBLE.

Le format de représentation en mémoire des matrices locales est décrit par la classe ARRAY2. Or les objets de type tableau sont tous dotés d'un attribut *area* référençant la zone mémoire où sont effectivement stockées les données du tableau. Pour que données d'une matrice locale puissent être émises ou reçues, il nous a donc suffi de définir dans la classe LOCAL_MATRIX les routines *send*, *recv_from*, etc. de manière à ce que les primitives de communication offertes par la classe POM soient invoquées avec en paramètre l'attribut *area*. Une partie de la classe LOCAL_MATRIX ainsi modifiée est reproduite dans l'exemple 2.7.

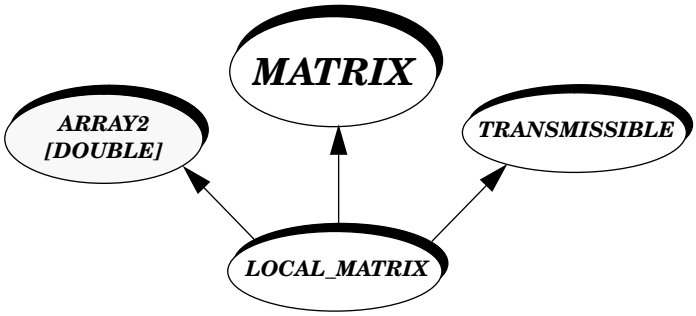


FIG. 2.4 - *Transformation des matrices locales en objets « transmissibles »*

Exemple 2.7

```
class LOCAL_MATRIX inherit
    MATRIX
        redefine nrow, ncolumn end;
    ARRAY2 [DOUBLE]
        rename height as nrow, width as ncolumn end;                    5
    TRANSMISSIBLE
creation
    make
feature {NONE} -- Interface with POM library
    POM: POM;                                                           10
feature -- Communication features
    send (destination: INTEGER) is
        do
            POM.send (destination, area);
        end; -- send                                                    15
    recv_from (source: INTEGER) is
        do
            POM.recv_from (source, area);
        end; -- recv_from
    ...                                                                 20
end -- LOCAL_MATRIX
```

Cas des vecteurs locaux

On a procédé d'une manière très similaire pour mettre en œuvre les vecteurs denses locaux dans Paladin. La classe LOCAL_VECTOR combine grâce à l'héritage multiple la spécification abstraite de la classe VECTOR, le format de représentation

interne décrit par la classe standard ARRAY, et les caractéristiques de transmissibilité de la classe TRANSMISSIBLE.

2.3.2 Autres formats envisageables pour les matrices locales

À ce jour, les classes de la bibliothèque Paladin ne permettent que la création et la manipulation de matrices et vecteurs denses. Toutefois, la bibliothèque pourrait aisément être enrichie de nouvelles classes décrivant d'autres types d'objets, tels que les matrices et vecteurs creux, les matrices symétriques ou triangulaires, etc. Enrichir la bibliothèque de nouvelles variantes de représentation se ramène simplement à ajouter de nouvelles classes dans la bibliothèque. En outre, grâce au mécanisme de l'héritage multiple, construire une classe décrivant un nouveau format de représentation interne s'avère particulièrement aisé : il suffit la plupart du temps de combiner la spécification abstraite de la classe MATRIX ou de la classe VECTOR avec les détails de mise en œuvre fournis par d'autres classes. On peut notamment mettre à profit les nombreuses classes standard de la bibliothèque Eiffel. Celles-ci décrivent des structures de données complexes telles que les listes, arbres, graphes, tables de hashage, etc. que l'on peut utiliser comme supports d'implantation pour représenter des matrices et vecteurs creux en mémoire.

On pourrait ainsi développer une nouvelle classe descendant de la classes abstraite MATRIX et décrivant un format de représentation interne pour les matrices creuses. Cette nouvelle classe — appelons la SPARSE_MATRIX (voir figure 2.5) — encapsulerait tous les détails relatifs à la représentation en mémoire d'une matrice creuse sous la forme, par exemple, d'une liste dynamique (la bibliothèque standard associée au langage Eiffel fournit un grand nombre de classes décrivant notamment différentes sortes de listes).

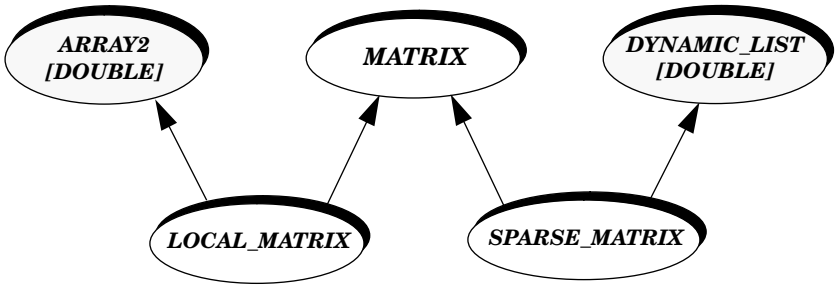


FIG. 2.5 - *Une mise en œuvre possible pour les matrices creuses*

Chapitre 3

Distribution des matrices dans Paladin

Nous avons *choisi* d'introduire dans un premier temps le parallélisme dans Paladin en procédant à la distribution des agrégats vecteurs et matrices. L'intégration d'un mécanisme de mémoire virtuelle partagée dans l'environnement EPEE est l'un des objectifs visés par Jean-Lin Pacherie dans le cadre de son travail de thèse. Avec ce mécanisme il sera possible de stocker les agrégats en mémoire partagée, et de baser la parallélisation sur la distribution du contrôle.

Dans le projet EPEE, l'un de nos objectifs fondamentaux est le maintien de la transparence de la distribution et du parallélisme qui en résulte vis à vis de l'utilisateur. Notre but est qu'à son niveau ne transparaisse de l'exécution sur APMD qu'un gain de performance — si possible proportionnel au nombre de processeurs employés — lorsqu'un agrégat distribué est utilisé sur une APMD dans le cadre d'un programme SPMD, à la place d'un agrégat non distribué équivalent utilisé sur une machine mono-processeur dans le cadre d'un programme séquentiel.

Idéalement, il faudrait que la distribution des agrégats soit totalement automatique. Cependant, de même que l'on ne sait pas encore très bien distribuer automatiquement les tableaux manipulés dans le cadre de programmes Fortran, il est en général difficile de mettre en œuvre des mécanismes capables de « choisir » automatiquement le schéma de distribution le plus approprié pour un vecteur ou une matrice dans Paladin. En général, ce schéma dépend en effet de nombreux facteurs, à commencer par la nature exacte des opérations devant être effectuées avec cet objet, et les caractéristiques des autres objets devant éventuellement intervenir lors de ces opérations.

Dans la pratique, on se contentera dans un premier temps d'une solution de compromis semblable à celle qui est adoptée dans les compilateurs-paralléliseurs

pour langages de type HPF : on demandera à l'utilisateur de spécifier explicitement les schémas de distribution des matrices qu'il souhaite utiliser, mais on ne lui demandera de gérer ni cette distribution, ni la parallélisation des calculs associés.

On montre dans ce chapitre comment, pour distribuer les matrices dans Paladin, on demande à l'utilisateur de spécifier explicitement les schémas de distribution requis pour ces objets. On montrera ensuite au chapitre 4 comment prendre en compte la distribution des matrices et vecteurs pour optimiser les calculs réalisés avec la bibliothèque Paladin. On verra enfin au chapitre 5 que la mise en œuvre de mécanismes de redistribution et de changement de format permet d'envisager le développement de bibliothèques d'agrégats dans lesquelles le choix des schémas de distribution des agrégats n'est plus à la charge du programmeur d'application.

3.1 Introduction

Ce chapitre décrit la mise en œuvre des matrices distribuées dans Paladin. Nous avons décomposé le problème de cette mise en œuvre en plusieurs sous-problèmes, que nous avons considérés et résolus séparément.

- Nous avons tout d'abord fait abstraction des problèmes de représentation interne des matrices distribuées, et développé des classes permettant de gérer des schémas de distribution pour les matrices. Ces classes sont décrites au paragraphe 3.2.

- Nous avons ensuite mis en œuvre des mécanismes permettant de maintenir l'interface séquentielle des matrices distribuées et garantissant du même coup la transparence de la distribution du point de vue de l'utilisateur. Ces mécanismes ont été encapsulés dans la classe DIST_MATRIX décrite au paragraphe 3.3.

- Nous avons enfin développé plusieurs classes concrètes proposant des formats de représentation interne alternatifs pour les matrices distribuées. Ces classes sont décrites au paragraphe 3.4.

Il va de soi que ces trois sous-problèmes ne sont pas *totalement* indépendants. Par exemple, le type de schéma de distribution considéré pour une matrice distribuée influence fatalement son format de représentation interne. Toutefois, on montre dans les paragraphes suivants que les mécanismes de la programmation par objets nous ont permis d'encapsuler dans des classes distinctes des « solutions » à chacun de ces sous-problèmes, et de combiner ensuite ces classes afin d'obtenir des classes concrètes décrivant des matrices distribuées conformément aux schémas choisis,

ayant une représentation interne adaptée à ces schémas, et présentant toutes une interface séquentielle à l'utilisateur.

3.2 Gestion de la distribution

3.2.1 Choix de schémas de distribution

Dans la version actuelle de Paladin, les schémas de distribution proposés pour les vecteurs et matrices sont très largement inspirés de ceux qu'autorise la syntaxe du langage HPF (*High Performance Fortran* [74]) pour les tableaux mono- et bi-dimensionnels. Ainsi, la distribution des matrices s'effectue sur la base d'un partitionnement en blocs homogènes, qui sont ensuite affectés aux nœuds de la machine parallèle cible.

Être capable de gérer la distribution d'une matrice signifie avant tout être capable, partant du couple d'indices (i, j) identifiant l'un des éléments scalaires de cette matrice, de déterminer l'identité du nœud propriétaire de cet élément ainsi que son adresse locale sur ce nœud. Gérer une matrice distribuée oblige donc à effectuer un très grand nombre de calculs élémentaires mais répétitifs. Les routines permettant d'effectuer ces calculs ont été encapsulées dans des classes distinctes. Ces classes, bien qu'ayant initialement été construites pour répondre aux besoins particuliers de Paladin, font à présent partie intégrante de la « boîte à outils » d'EPEE. En effet, leur champs d'application n'est pas limité à la seule distribution des vecteurs et matrices. Elles peuvent notamment servir de briques de base pour construire des classes capables de gérer des tableaux à K dimensions.

Le problème de la gestion de la distribution des matrices peut être décomposé en deux sous-problèmes : la gestion du partitionnement, et la gestion du placement. Ces sous-problèmes sont abordés successivement dans les paragraphes suivants.

3.2.2 Gestion du partitionnement

Le partitionnement d'une matrice est décrit par un couple de paramètres (bfi, bfj). Ces paramètres, baptisés *facteurs de partitionnement*, servent à fixer la taille des blocs. Les blocs d'une matrice distribuées sont donc en principe tous de taille $bfi \times bfj$. Cependant, on n'impose pas dans Paladin que les facteurs bfi et bfj soient des diviseurs des dimensions m et n de la matrice considérée. Il peut donc arriver que certains blocs soient éventuellement plus petits que la taille spécifiée : il s'agit alors de blocs couvrant le « bord droit » ou le « bord inférieur » de la matrice considérée, comme illustré dans la figure 3.1.

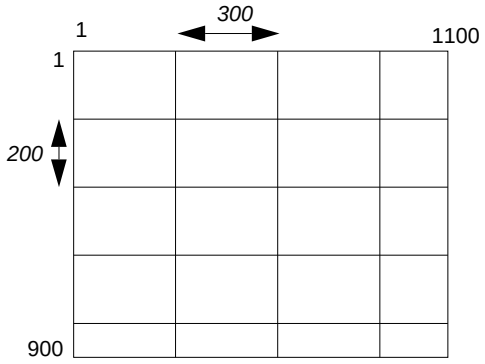


FIG. 3.1 - *Partitionnement d'une matrice de taille* 900×1100 *en blocs de taille* 200×300

Chaque bloc est identifié par un couple d'indices (bi, bj), défini sur le domaine d'indices $[0..bimax, 0..bjmax]$ (voir la figure 3.2). Pour tout couple d'indices (i, j) défini sur le domaine d'indices global (celui de la matrice considérée), on doit être capable de déterminer l'identité (bi, bj) du bloc englobant.

On peut aisément calculer (bi, bj) en fonction de (i, j) et des facteurs de partitionnement (bfi, bfj) :

$$\begin{aligned} bi(i, bfi) &= (i-1) \textbf{ div } bfi \\ bj(j, bfj) &= (j-1) \textbf{ div } bfj \end{aligned}$$

Dans la version actuelle de Paladin, on a décidé de préserver la structure bi-dimensionnelle des blocs. Les coordonnées locales (li, lj) de l'élément (i, j) dans le bloc (bi, bj) peuvent donc être exprimées en fonction de (i, j) et de (bfi, bfj) :

$$\begin{aligned} li(i, bfi) &= (i-1) \textbf{ mod } bfi + 1 \\ lj(j, bfj) &= (j-1) \textbf{ mod } bfj + 1 \end{aligned}$$

Le fait de considérer les blocs comme des structures bi-dimensionnelles résulte clairement d'un choix, motivé par le fait que nous avons décidé de mettre en œuvre et de manipuler ces blocs comme étant des objets de type matrice (cet aspect de la mise en œuvre des matrices distribuées est abordé plus en détails dans le paragraphe 3.4). D'autres approches seraient tout aussi envisageables. On pourrait par exemple adopter la même technique de distribution que celle utilisée dans le compilateur-parallèliseur Pandore. Dans cet outil, la distribution d'un tableau passe par une phase de linéarisation de ce tableau, suivie d'une pagination du domaine mono-dimensionnel ainsi obtenu. Les pages sont ensuite réparties sur l'ensemble des nœuds disponibles [94, 93]. Nous aurions pu choisir de distribuer ainsi les matrices

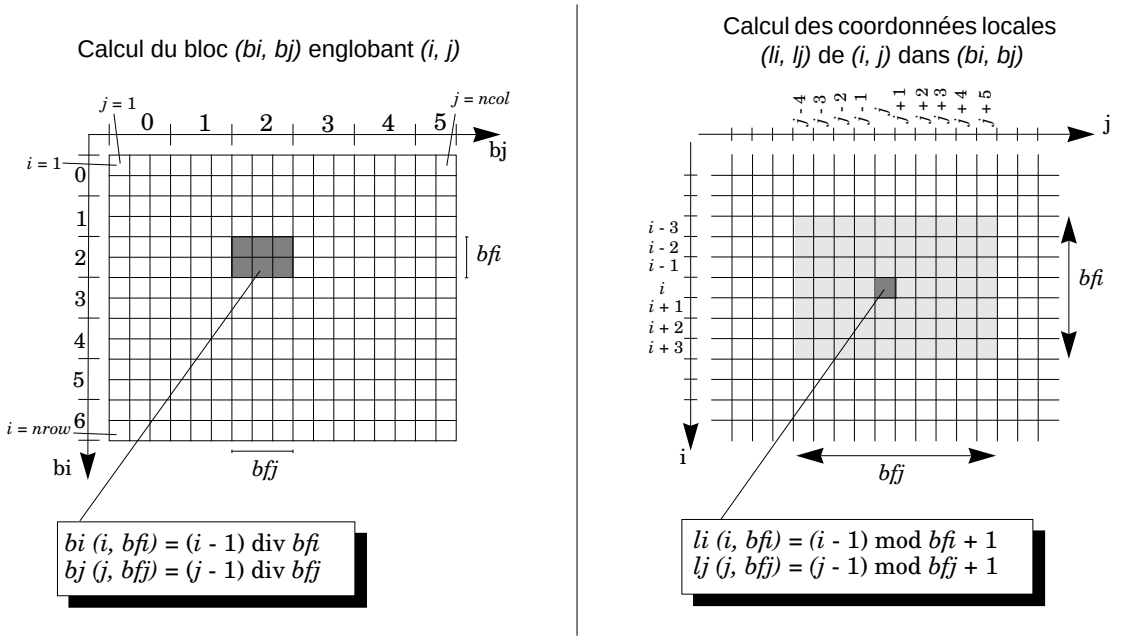


FIG. 3.2 - *Localisation d'un élément (i, j) par identification du bloc englobant (à gauche) et calcul des coordonnées locales dans ce bloc (à droite)*

dans Paladin. Toutefois, cette approche ne nous aurait probablement pas permis d'appliquer de manière aussi systématique le principe de la réutilisation de code. On verra en effet au paragraphe 3.4 qu'avec l'approche retenue dans Paladin, chaque bloc résultant du partitionnement d'une matrice peut lui-même être mis en œuvre sous la forme d'un objet de type matrice (ou, dans certains cas, de type vecteur).

La classe DISTRIBUTION_2D

Les procédures et fonctions permettant de gérer la distribution d'une matrice ont été encapsulées dans la classe DISTRIBUTION_2D. Parmi les services offerts par cette classe, on peut citer des fonctions servant à localiser un élément scalaire donné, voire un bloc de partitionnement donné (*owner_of_item* et *owner_of_block*), et à effectuer les conversions d'indices requises lorsqu'il faut, connaissant les coordonnées globales (i, j) d'un élément, déterminer les coordonnées locales (li, lj) correspondantes dans le bloc englobant.

On a reproduit dans l'exemple 3.1 une partie de l'interface de la classe DISTRIBUTION_2D.

Le paramétrage de la routine de création *make* de la classe DISTRIBUTION_2D est inspiré de la syntaxe utilisée dans la directive DISTRIBUTE du langage HPF. La

Exemple 3.1

```
class interface DISTRIBUTION_2D
    ...
creation
    make

feature -- Creation
    make (rows, columns, bfi, bfj: INTEGER; mapping: MAPPING_2D)

feature -- Block location
    owner_of_block (bi, bj: INTEGER): INTEGER
        -- Identification of processor supporting block(bi, bj)
    ...
feature -- Element location
    owner_of_item (i, j: INTEGER): INTEGER
        -- Identification of processor supporting item(i, j)
    ...
feature -- Index conversion
    nbi (i: INTEGER): INTEGER
    nbj (j: INTEGER): INTEGER
        -- (bi, bj):= Identification of block supporting item(i, j)

    lbi (i: INTEGER): INTEGER
    lbj (j: INTEGER): INTEGER
        -- (li, lj):= Local identification of item(i, j)
    ...
end -- interface DISTRIBUTION_2D
```

classe DISTRIBUTION_2D a d'ailleurs à peu près la même puissance d'expression que le langage HPF lorsqu'il s'agit de distribuer des structures bi-dimensionnelles.

Le programmeur d'application décrit un schéma de distribution en spécifiant la taille du domaine d'indices considéré (nombre de lignes et de colonnes), la taille des blocs de partitionnement (facteurs *bfi* et *bfj*), et en indiquant quelle fonction de placement doit être utilisée pour affecter les blocs aux nœuds de la machine parallèle cible (le problème du placement des blocs est abordé dans le paragraphe 3.2.3).

La classe DISTRIBUTION_2D a en fait été construite en mettant à profit le mécanisme de l'héritage multiple : la classe DISTRIBUTION_2D hérite deux fois des mécanismes encapsulés dans la classe DISTRIBUTION_1D (voir figure 3.3), qui est dédiée au partitionnement et à la distribution des structures mono-dimensionnelles telles que vecteurs, listes fixes, etc. Cet exemple montre bien l'avantage de l'héritage multiple par rapport à l'héritage simple : de nouvelles classes peuvent aisément être construites en combinant simplement les caractéristiques héritées de plusieurs classes parentes, et éventuellement en héritant plusieurs fois d'une même classe parente. Ainsi, une classe dédiée à la gestion de la distribution de structures à trois dimensions pourrait également être construite à l'aide de l'héritage multiple, soit en héritant une fois de la classe DISTRIBUTION_2D et une fois de la classe DISTRIBUTION_1D (comme illustré dans la figure 3.3), soit en héritant trois fois de DISTRIBUTION_1D.

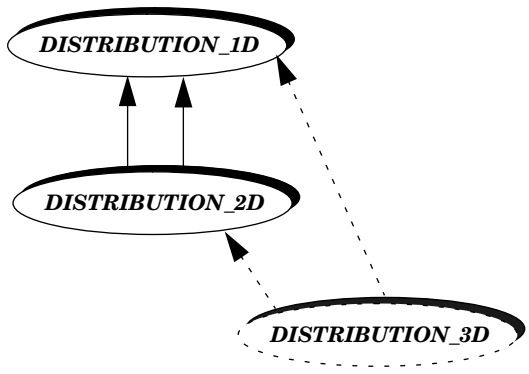


FIG. 3.3 - *Structure d'héritage des classes gérant la distribution*

3.2.3 Gestion du placement

Le placement consiste à affecter chaque bloc résultant du partitionnement à un nœud de la machine cible. La bibliothèque POM associée à l'environnement EPEE

fournit l'abstraction d'une machine parallèle virtuelle dotée de N nœuds, numérotés de 0 à $N-1$. À chaque bloc (bi, bj) on doit donc faire correspondre un numéro de nœud pris dans l'intervalle $[0..N-1]$.

La classe différée MAPPING_2D dont l'interface est reproduite dans l'exemple 3.2 contient la déclaration d'une fonction de placement, baptisée *map_block*. Cette fonction retourne l'identité du nœud propriétaire du bloc (bi, bj) considéré. Elle est paramétrée par :

- le couple (bi, bj) qui identifie le bloc dont on doit calculer le possesseur ;
- le couple (bi_{max}, bj_{max}) caractérisant la taille du domaine d'indices considéré ;
- le nombre P de nœuds disponibles dans la machine cible.

Exemple 3.2

```
deferred class interface MAPPING_2D
feature
    map_block (bi, bj, bimax, bjmax, P : INTEGER) : INTEGER
            -- Maps block(bi, bj) on a processor whose identifier
            -- must be in the range [0, P-1]                          5
        require
            bi_valid : (bi >= 0) and (bi < bimax)
            bj_valid : (bj >= 0) and (bj < bjmax)
        deferred
        ensure                                                        10
            (Result >= 0) and (Result < P)
end -- class interface MAPPING_2D
```

La définition de la fonction *map_block* a volontairement été maintenue à l'extérieur de la classe DISTRIBUTION_2D, afin que tout utilisateur de Paladin puisse s'il le désire définir une fonction de placement en fonction de ses besoins propres. On peut en effet développer une multitude de classes descendant de MAPPING_2D, chacune de ces classes encapsulant une mise en œuvre particulière de la fonction *map_block*.

À ce jour, nous n'avons intégré à la bibliothèque Paladin que deux classes concrètes descendant de MAPPING_2D et permettant de réaliser un placement des blocs sur un ensemble de nœuds, soit « dans le sens des lignes », soit « dans le sens des colonnes » (voir figure 3.4).

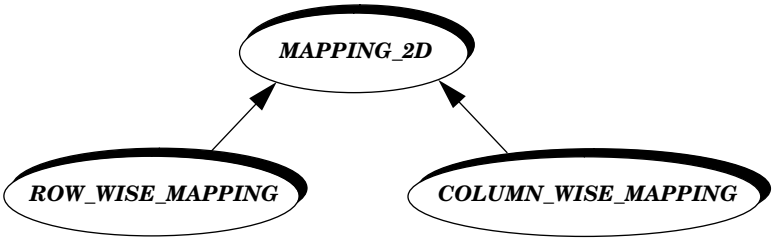


FIG. 3.4 - *Structure d'héritage des classes décrivant le placement*

Exemple 3.3

```
expanded class ROW_WISE_MAPPING
inherit MAPPING_2D
feature
    map_block (bi, bj, bimax, bjmax, P: INTEGER): INTEGER is
        do                                                          5
            Result := (bi * bjmax + bj) \\ P;
        end;
end -- class ROW_WISE_MAPPING
```

Dans la fonction *map_block* de la classe ROW_WISE_MAPPING, on procède à une linéarisation (dans le sens des lignes) du domaine bi-dimensionnel $[0 : bi_{max} - 1, 0 : bj_{max} - 1]$ et l'on replie ensuite le domaine mono-dimensionnel qui en résulte $[0 : bi_{max} \times bj_{max} - 1]$ sur le domaine $[0 : P - 1]$ correspondant à l'ensemble des nœuds disponibles sur la machine cible.

La fonction *map_block* de la classe COLUMN_WISE_MAPPING encapsule une équation similaire, si ce n'est que la linéarisation est réalisée dans le sens des colonnes.

On a illustré dans la figure 3.5 les deux types de placement pouvant être obtenus selon que l'on utilise la fonction de la classe ROW_WISE_MAPPING (dessin de gauche) ou celle de la classe COLUMN_WISE_MAPPING (dessin de droite). Dans cet exemple, on suppose que le placement doit être réalisé sur une machine offrant 4 nœuds.

Extensibilité du mécanisme de placement

Les schémas de placement autorisés par l'emploi des classes ROW_WISE_MAPPING et COLUMN_WISE_MAPPING n'ont pas d'équivalent dans le langage HPF. Le placement est réalisé dans HPF dimension par dimension, en projetant le tableau

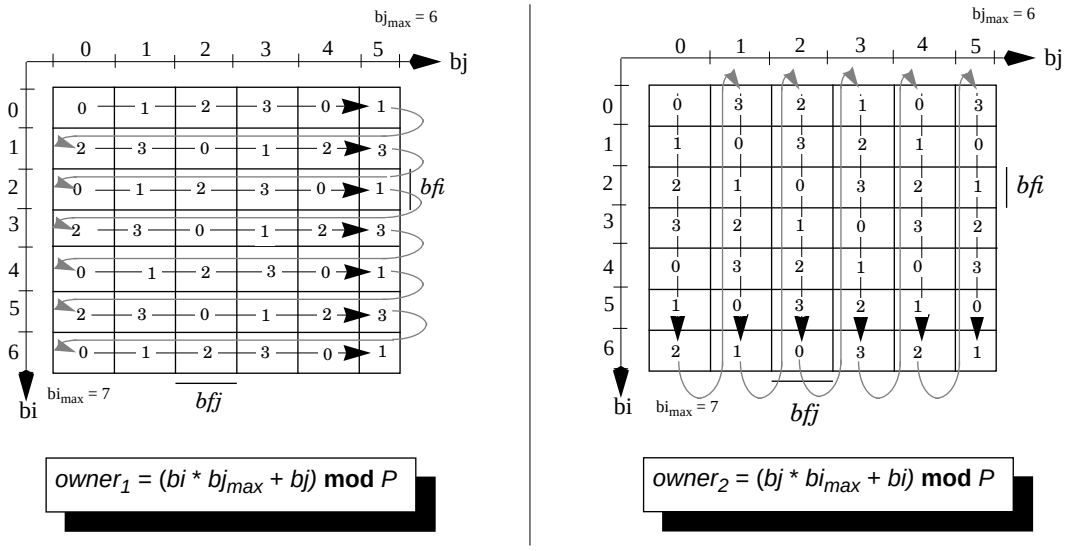


FIG. 3.5 - *Placement des blocs réalisé dans le sens des lignes (à gauche) et dans le sens des colonnes (à droite) sur un ensemble de 4 nœuds.*

à N dimensions résultant du partitionnement[1] sur un tableau de nœuds ayant également N dimensions. Il n'y a donc pas dans HPF de phase de linéarisation comme c'est le cas avec les deux classes évoquées plus haut. Sur ce point, les schémas de placement mis en œuvre dans Paladin s'apparentent plutôt aux schémas autorisés par le compilateur Pandore.

Il serait possible de construire de nouvelles classes descendant de MAPPING_2D et d'y implanter la fonction *map_block* en s'inspirant des schémas de placement autorisés dans HPF. En fait, un programmeur imaginatif pourrait certainement proposer des politiques de placement plus exotiques, telles qu'un placement aléatoire, ou bien encore un placement réalisé « dans le sens des diagonales ». Pour expérimenter une nouvelle politique de placement dans Paladin, il suffit de construire une classe descendant de MAPPING_2D, et d'y encapsuler une mise en œuvre appropriée de la routine *map_block*.

3.2.4 Notion de descripteur de distribution

Dans Paladin, chaque matrice distribuée doit être associée lors de sa création à une instance de la classe DISTRIBUTION_2D faisant office de descripteur de distribution (on parle de *template* dans la terminologie HPF) pour cette matrice. Une

1. Dans le cas d'un schéma de distribution décrit à l'aide de la directive HPF DISTRIBUTE.

matrice pourra être redistribuée en changeant de descripteur dynamiquement, et plusieurs matrices pourront « partager » un même schéma de distribution en référençant le même descripteur.

Exemple 3.4

```
local
    my_dist : DISTRIBUTION_2D ;
    my_mapping : COLUMN_WISE_MAPPING ;
do
  !!my_dist.make (10, 10, 5, 2, my_mapping) ;                    5
    ...
end ;
```

L'exemple 3.4 illustre la création d'une instance de la classe DISTRIBUTION_2D. La routine de création prend en paramètres la taille du domaine d'indices considéré, les paramètres de partitionnement, et une entité référençant un objet dont le type est conforme à MAPPING_2D. L'instance de la classe DISTRIBUTION_2D ainsi créée permettra de gérer la distribution d'un domaine d'indices de taille 10×10, partitionné en blocs de taille 5×2 qui devront être affectés dans le sens des colonnes aux nœuds de la machine cible.

On pourra noter que dans l'exemple 3.4, on n'instancie pas explicitement l'objet *my_mapping*. Ceci est dû au fait que la classe ROW_WISE_MAPPING est une classe expansée (voir la ligne 1 de l'exemple 3.3, ainsi que le point de langage 3.1), ce qui signifie que toute entité déclarée comme étant de type ROW_WISE_MAPPING référence directement un objet de ce type.

Note sur la réification de la fonction de placement

La classe MAPPING_2D et ses descendantes ne décrivent pas un type de donnée mais un type de fonction : ce sont des *abstractions algorithmiques*. Les objets obtenus par instanciation de COLUMN_WISE_MAPPING ou de ROW_WISE_MAPPING ne sont donc pas des objets ordinaires : ce sont des objets *agents*.

Le concept d'objet agent a initialement été introduit par C. Hewitt dans [3] pour désigner des entités actives, et repris ensuite par G. Booch dans [24] pour désigner des « abstractions algorithmiques ». Selon G. Booch, on doit en effet distinguer en programmation par objets entre les classes décrivant des structures (abstractions structurelles) comme par exemple les listes, les tableaux, les arbres, etc., et les classes décrivant des *agents* fournissant des services d'ordre algorithmique (abstractions algorithmiques), comme par exemple les curseurs, les itérateurs, etc.

Point de langage 3.1

Classe expansée. Lorsqu'une classe C est déclarée avec le mot-clé **expanded**, il n'est pas nécessaire de créer explicitement les instances de cette classe. Tout attribut déclaré comme étant de type C dans un programme d'application peut directement être manipulé comme un objet de ce type.

La mise en œuvre d'objets agents permet de combler l'une des « lacunes » apparentes des langages à objets à typage statique : le fait que dans de tels langages les fonctions ne sont pas des objets de première classe (*i.e.* pouvant être passées en paramètre à une procédure[2]).

3.2.5 Perspectives d'extension

Alors que le développement de compilateurs-paralléliseurs pour langages de type HPF impose en général que des mécanismes de gestion de la distribution soient spécifiés une fois pour toutes avant d'être encapsulés dans un outil de parallélisation, l'approche illustrée dans Paladin laisse le champs libre à toute perspective d'extension. On a énuméré ci-dessous quelques unes des extensions envisageables.

- **Distribution de tableaux à N dimensions** : de même qu'on a construit la classe DISTRIBUTION_2D en héritant deux fois de la classe DISTRIBUTION_1D, on pourrait en s'appuyant sur le mécanisme de l'héritage multiple développer de nouvelles classes capables de gérer la distribution de tableaux à N dimensions (voir la figure 3.1).

- **Partitionnement hétérogène** : les classes qui gèrent la distribution des matrices dans Paladin ont été construites afin de gérer des schémas de distribution « à la HPF », les contraintes de partitionnement d'une matrice étant décrites par un couple de facteurs de partitionnement (bfi, bfj). Le partitionnement est donc homogène : les blocs sont tous de taille identique. On pourrait cependant développer des classes descendant de DISTRIBUTION_2D permettant de gérer des matrices partitionnées de manière hétérogène. Ce type de partitionnement serait certainement très utile pour gérer par exemple certaines formes de matrices creuses distribuées (en l'occurrence les matrices creuses dont les éléments non nuls ne sont pas répartis uniformément, mais sont au contraire regroupés en amas).

2. Le langage C++ permet de passer des fonctions en paramètres, mais il s'agit là de l'une des nombreuses caractéristiques qu C++ a « hérité » du langage C et qui lui doivent d'ailleurs d'être souvent qualifié de langage « hybride ».

- **Alignement entre descripteurs** : dans le langage HPF, on peut décrire la distribution d'un *template*, soit de manière explicite en utilisant la directive DISTRIBUTE, soit de manière implicite en spécifiant son positionnement par rapport à un autre *template* à l'aide de la directive ALIGN. On pourrait de même aligner les uns par rapport aux autres les descripteurs de distribution manipulés dans Paladin (dont on rappelle qu'ils jouent approximativement le même rôle que les *templates* de HPF). Pour ce faire, on pourrait par exemple procéder de manière à peu près similaire à celle qui a permis de construire les « vues » sur les matrices et vecteurs. Un descripteur de distribution serait alors, soit un objet capable de procéder effectivement aux calculs requis pour gérer un schéma de distribution donné, soit une « vue » paramétrée par une référence à un descripteur pré-existant et par des informations additionnelles de positionnement (décalage, rotation, etc.).

3.3 Abstraction des matrices distribuées

Ayant implanté dans les classes DISTRIBUTION_2D et MAPPING_2D tous les mécanismes nécessaires à la *gestion* des schémas de distribution choisis pour les matrices distribuées, nous avons ensuite encapsulé dans la classe DIST_MATRIX tous les détails relatifs à la distribution effective des matrices conformément à ces schémas. Cette classe est partiellement reproduite dans l'exemple 3.5.

On notera que :

- la classe DIST_MATRIX hérite de la spécification abstraite de la classe MATRIX (ligne 2) ;
- elle est dotée de plusieurs routines de création (lignes 5 à 8). Le schéma de distribution d'une matrice peut ainsi être spécifié lors de sa création, soit de manière explicite — dans ce cas un nouveau descripteur de distribution est automatiquement créé et associé à la matrice —, soit de manière implicite en référençant une matrice distribuée pré-existante ou un descripteur de distribution pré-existant ;
- le descripteur de distribution auquel la matrice courante est associée dès sa création est ensuite référencé au sein de cette matrice par l'attribut *dist* (ligne 11) ;
- chaque matrice distribuée peut accéder aux divers services offerts par la classe POM [3] par l'intermédiaire de l'attribut *POM* (ligne 14). La clause d'exportation sélective « **feature** {*NONE*} » à laquelle est soumis l'attribut *POM*

3. La classe POM a été présentée au § 1.2.4.

Exemple 3.5

```
deferred class DIST_MATRIX inherit
   MATRIX   -- Abstract specification

feature -- Creation
   make (rows, cols, bfi, bfj: INTEGER; alignment: MAPPING_2D) is
      deferred end;
   make_from (new_dist: DISTRIBUTION_2D) is  deferred end;
   make_like (other: DIST_MATRIX) is deferred end;

feature -- Distribution template
   dist: DISTRIBUTION_2D;

feature {NONE}  -- Communication features
   POM: POM;

feature -- Accessors
   item (i, j: INTEGER): like item is do ... end;
   put (v: like item; i, j: INTEGER) is do ... end;
end -- class DIST_MATRIX
```

(ligne 13) spécifie que cet attribut n'est pas destiné à être utilisé depuis une classe cliente de la classe DIST_MATRIX. Cette clause d'exportation fait de l'attribut *POM* un attribut *privé*. En outre, la classe POM étant une classe expansée (voir l'exemple 1.1 page 25 et le point de langage 3.1 page 74), on n'a pas à créer explicitement d'objet de type POM pour l'affecter à l'attribut *POM*.

Mise en œuvre des accesseurs de base

Nous avons mis en œuvre l'accesseur *put* dans la classe DIST_MATRIX de manière à garantir le respect de la règle dite « des écritures locales » (ou *Owner Write Rule*). Cette règle a été introduite au paragraphe 1.3.3.3. Elle spécifie que seul le nœud qui possède une partition issue d'un agrégat distribué peut en modifier le contenu.

Appliquée aux matrices distribuées, la règle des écritures locales stipule que seul le nœud propriétaire de l'élément scalaire (i, j) d'une matrice distribuée peut en modifier la valeur. Lorsqu'un programme d'application SPMD contient une expression de la forme *M.put(v, i, j)* – *M* étant une matrice distribuée – seul le nœud propriétaire de l'élément (i, j) est donc en mesure d'effectuer l'affectation demandée.

Dans la mise en œuvre de l'accesseur *put*, nous avons conditionné l'opération d'écriture par un test de localité, comme illustré dans l'exemple 3.6. On pourra noter l'invocation de la fonction de localisation *item_is_local* sur le descripteur de distribution associé à la matrice courante.

Exemple 3.6

```
put (v: like item; i, j: INTEGER) is
    do
        if dist.item_is_local(i, j) then
            local_put (v, i, j)
        end; -- if                                          5
    end;
```

Nous avons également implanté l'accesseur *item* dans la classe DIST_MATRIX de manière à préserver l'interface séquentielle des matrices distribuées : lorsqu'un programme d'application SPMD contient une expression de la forme *v := M.item(i, j)*, la fonction *item* doit impérativement retourner la même valeur sur tous les nœuds participant au calcul. Dans la mise en œuvre de *item*, nous avons donc fait en sorte que le nœud propriétaire de l'élément (i, j) en diffuse la valeur à l'intention de tous les autres nœuds (voir l'exemple 3.7).

Les mécanismes mis en œuvre dans la classe DIST_MATRIX afin d'assurer, d'une part le respect de la règle des écritures locales, et d'autre part les accès distants, s'ap-

Exemple 3.7

```
item (i, j: INTEGER): DOUBLE is
    do
        if dist.item_is_local(i, j) then
            Result := local_item (i, j);
            POM.bcast (Result);                                          5
        else
            POM.recv_bcast_from (dist.owner_of_item (i, j), Result);
        end; -- if
    end;
```

parentent aux mécanismes de l'*Exec* et du *Refresh* introduits au paragraphe 1.3.3.3. Le mécanisme de l'*Exec* garantit un accès en écriture sur le seul nœud propriétaire de la donnée considérée, alors que le mécanisme du *Refresh* assure la mise à disposition de tous les nœuds d'une copie d'une donnée accédée en lecture (ce phénomène est perçu comme un « rafraîchissement » de la donnée sur tous les nœuds).

Rôle et mise en œuvre des accesseurs locaux

Dans les exemples 3.6 et 3.7, on a vu apparaître deux nouvelles routines, *local_put* et *local_item*, qui n'ont pas encore été évoquées : ces deux routines sont des accesseurs locaux, dont la mise en œuvre dépend étroitement du format choisi pour représenter sur chaque nœud les blocs dont il est propriétaire. Ces accesseurs locaux n'ont donc pas été mis en œuvre dans la classe DIST_MATRIX : nous les y avons simplement déclarés et maintenus différés de manière à ce qu'ils puissent être définis dans les classes descendant de DIST_MATRIX (voir l'exemple 3.8).

La classe DIST_MATRIX contenant des routines différées, elle n'est pas pleinement opérationnelle : dans cette classe on se contente de résoudre le problème des accès aux données locales et distantes, sans essayer cependant de résoudre celui de leur représentation en mémoire. De même que la classe MATRIX décrit les entités matrices en faisant totale abstraction de leur mise en œuvre, la classe DIST_MATRIX décrit les entités matrices distribuées en faisant totale abstraction de leur format de représentation interne.

Primitives « locales » et primitives « SPMD »

Les accesseurs *local_put* et *local_item* étant des accesseurs *locaux*, ils ne peuvent être invoqués que dans le cadre d'une commande gardée exécutée par le seul nœud propriétaire de l'élément dont on cherche à écrire ou à lire la valeur. Dans les

Exemple 3.8

```
deferred class DIST_MATRIX
    ...
feature {PALADIN} -- Local accessors
    local_put (v: like item; i, j: INTEGER) is
        require                                                    5
            valid_item: local_item (i, j);
        deferred end;
    local_item (i, j: INTEGER): like item is
        require
            valid_item: local_item (i, j);                         10
        deferred end;
end -- class DIST_MATRIX
```

paragraphes et chapitres suivants, on décrira d'autres *routines locales* que nous avons implantées dans Paladin. Comme les accesseurs *local_put* et *local_item*, ces routines porteront toutes un nom pourvu du préfixe *local_*. Ce préfixe a valeur d'avertissement : il attire l'attention du lecteur sur le fait que la routine considérée n'a été conçue qu'afin de servir à la gestion *interne* d'un agrégat distribué, et que lors de son exécution un nœud ne procède qu'à des opérations locales (ce qui exclut notamment toute possibilité d'interaction entre ce nœud et d'autres nœuds).

Les routines locales implantées dans Paladin n'étant aucunement destinées à être invoquées au niveau d'un programme d'application SPMD, elles ont toutes été rendues « invisibles » pour le programmeur d'application grâce au mécanisme de l'exportation sélective. Dans le cas des accesseurs *local_put* et *local_item*, par exemple, on peut constater dans l'exemple 3.8 que la clause d'exportation de ces routines précise qu'elles ne doivent être perceptibles qu'au niveau des classes héritant de la classe PALADIN (voir figure 3.6). On assure ainsi que les routines locales ne pourront être utilisées que par un programmeur participant à l'extension de la bibliothèque. En revanche, pour l'utilisateur de la bibliothèque (qui n'utilise les classes de Paladin qu'en tant que *client* pour bâtir des programmes d'application), les routines locales demeurent totalement invisibles. On pourra en outre noter dans l'exemple 3.8 la présence de préconditions dans les déclarations de *local_put* et *local_item*. Ces préconditions garantissent le respect des accès locaux : seul le nœud propriétaire d'un élément (i, j) donné peut y accéder localement.

Contrairement aux accesseurs *local_put* et *local_item*, les accesseurs *put* et *item* ont été spécialement implantés dans la classe DIST_MATRIX de manière à pouvoir être invoqués sur une matrice distribuée dans le contexte d'exécution d'un programme SPMD. Ces accesseurs sont devenus des *accesseurs SPMD*. Plus géné-

ralement, on appellera *routine SPMD* une routine pouvant être invoquée sur un agrégat distribué au niveau d'un programme SPMD, et entraînant de la part de cet agrégat le même « comportement » qu'on observerait de la part d'un agrégat non distribué équivalent dans un contexte d'exécution séquentiel.

En fait, après avoir été mis en œuvre comme montré dans les exemples 3.6 et 3.7, les accesseurs *put* et *item* de la classe DIST_MATRIX suffisent à garantir de la part d'une matrice distribuée le même « comportement » que celui d'une matrice non distribuée. Les autres accesseurs ainsi que les opérateurs permettant de manipuler une matrice ont tous été dotés dans la classe MATRIX d'une mise en œuvre par défaut s'appuyant sur les accesseurs de base *put* et *item*. Cette mise en œuvre est totalement indépendante de la représentation interne — et notamment de la distribution éventuelle — de la matrice sur laquelle ces routines peuvent être invoquées. Ces routines peuvent donc être invoquées sur une matrice distribuée au niveau d'un programme SPMD : ce sont des routines SPMD.

3.4 Représentation interne des matrices distribuées

Nous avons intégré dans la bibliothèque Paladin plusieurs classes décrivant divers formats de représentation interne pour les matrices distribuées.

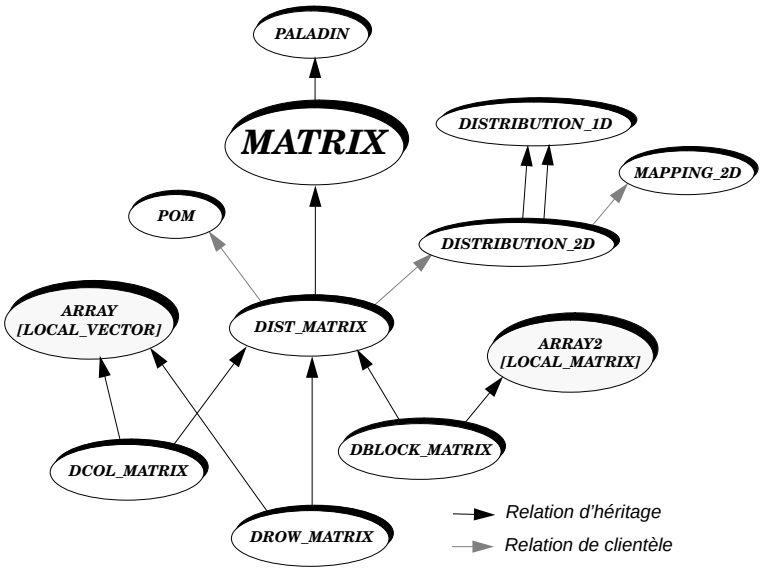


FIG. 3.6 - *Intégration des matrices distribuées dans Paladin*

3.4.1 Matrices distribuées par blocs

3.4.1.1 Principe

La classe DBLOCK_MATRIX hérite à la fois de la classe DIST_MATRIX et de la classe ARRAY2[LOCAL_MATRIX] (voir l'exemple 3.9 et la figure 3.6). Elle met en œuvre une matrice distribuée par blocs sous la forme d'une table bi-dimensionnelle de blocs matrices, chaque bloc matrice étant lui même représenté sous la forme d'une instance de la classe LOCAL_MATRIX.

Exemple 3.9

```
class DBLOCK_MATRIX inherit
    DIST_MATRIX
    ARRAY2 [LOCAL_MATRIX]
        rename
            make as make_table,                                    5
            put as put_local_block, item as local_block
        end
    ...
end -- class DBLOCK_MATRIX
```

3.4.1.2 Initialisation de la table des blocs

La table des blocs matrices est créée et initialisée lors de la création d'un objet de type DBLOCK_MATRIX. On a reproduit dans l'exemple 3.10 l'une des routines de création de la classe DBLOCK_MATRIX. On notera dans cet exemple que l'attribut *dist* référençant le descripteur de distribution de la matrice courante permet de disposer pour cette matrice de toutes les fonctions de gestion de la distribution définies dans la classe DISTRIBUTION_2D (fonctions de localisation, de conversion d'indices, etc.).

Exemple 3.10

```
make_from (new_dist : DISTRIBUTION_2D) is
        -- Build table of local blocks
    local
      bi, bj : INTEGER ; new_block : like local_block ;
    do                                                          5
        -- Adopt distribution descriptor
        dist := new_dist ;

        -- Create the table of blocks
        make_table (dist.nbimax, dist.nbjmax);                  10

        -- Fills in the table of blocks
        from bi := 0 until bi > dist.nbimax loop
            from bj := 0 until bj > dist.nbjmax loop
                if dist.block_is_local (bi, bj) then            15
                   !!new_block.make (dist.lbimax(bi), dist.lbjmax(bj));
                    put_local_block (new_block, bi, bj);
                end ; -- if
                bj := bj + 1 ;
            end ; -- loop                                       20
            bi := bi + 1 ;
        end ; -- loop
    end ;
```

La procédure d'initialisation consiste à créer sur chaque nœud une instance de la classe LOCAL_MATRIX pour chacun des blocs (bi, bj) appartenant à ce nœud. On affecte ensuite le bloc matrice ainsi créé à l'entrée (bi, bj) de la table des blocs. À l'issue de la phase d'initialisation, chaque entrée non vide dans la table des blocs référence un objet de type LOCAL_MATRIX. Une entrée vide dans la table (une lecture de cette entrée retourne la valeur prédéfinie **Void**) indique que le nœud local n'est pas propriétaire du bloc considéré.

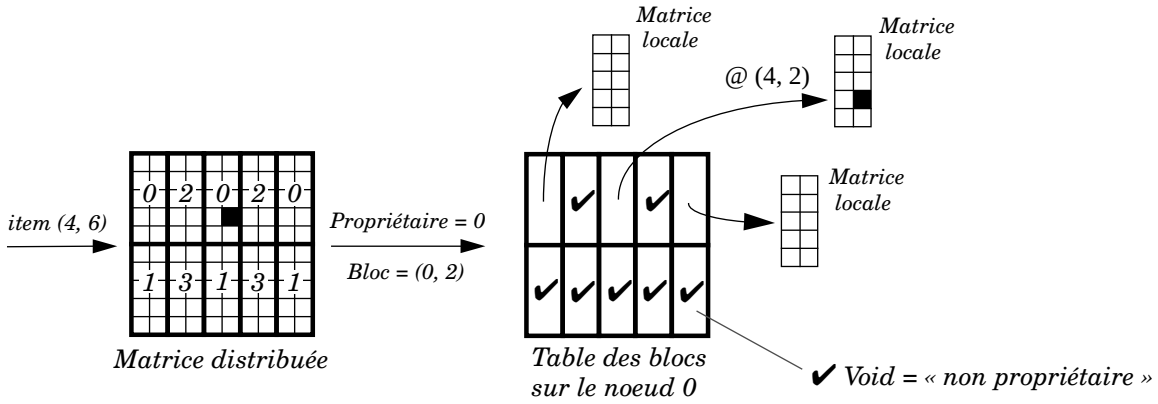


FIG. 3.7 - *Représentation interne d'une matrice distribuée par blocs et illustration du mécanisme d'accès à un élément scalaire*

La figure 3.7 montre la représentation interne sous forme de table de blocs matrices d'une matrice de taille 10×10 partitionnée en blocs de taille 5×2. Pour chaque bloc résultant du partitionnement on a indiqué (dans la partie gauche de la figure) l'identité du nœud qui en est propriétaire, en supposant que la matrice considérée est ici distribuée sur 4 nœuds. En partie droite on a schématisé le contenu de la table des blocs sur le nœud 0. Chaque entrée de la table contient soit la valeur prédéfinie **Void**, soit une référence vers un objet de type LOCAL_MATRIX.

La figure montre également comment, partant des coordonnées globales d'un élément scalaire quelconque, on peut identifier le bloc matrice englobant et les coordonnées locales de l'élément dans ce bloc. Ainsi, à l'élément de coordonnées globales $(4, 6)$ correspond l'élément de coordonnées locales $(4, 2)$ dans le bloc matrice $(0, 2)$ appartenant au nœud 0.

3.4.1.3 Mise en œuvre des accesseurs locaux de base

Les accesseurs locaux *local_put* et *local_item*, que nous n'avons pu définir au niveau de la classe DIST_MATRIX, doivent être définis ici de manière à tenir compte

de l'indirection due à la table des blocs. Pour accéder à un élément (i, j) donné, on doit d'abord localiser le bloc matrice (bi, bj) englobant, puis accéder ensuite dans ce bloc matrice en lecture ou en écriture à l'élément local (li, lj) correspondant à l'élément global (i, j) désiré. Dans la classe DBLOCK_MATRIX, les accesseurs locaux *local_put* et *local_item* sont donc définis comme illustré dans l'exemple 3.11.

Exemple 3.11

```
local_item (i, j: INTEGER): like item is
    do
        Result := local_block (dist.nbi(i), dist.nbj(j)).item (dist.lbi(i), dist.lbj(j));
    end;

local_put (v: like item; i, j: INTEGER) is
    do
      local_block (dist.nbi(i), dist.nbj(j)).put (v, dist.lbi(i), dist.lbj(j));
    end;
```

3.4.1.4 Mécanismes pour les transferts de blocs

Dans la classe DBLOCK_MATRIX, la représentation interne sous forme de table de blocs matrices nous a incité à implanter des mécanismes permettant de transférer directement des blocs matrices entre les nœuds. Ces mécanismes ne sont pas destinés à être utilisés directement par un utilisateur dans un programme d'application. Ils constituent en revanche un atout majeur pour l'optimisation interne de la classe DBLOCK_MATRIX. On montrera dans le chapitre 4 comment certains des opérateurs hérités de MATRIX ont été redéfinis dans la classe DBLOCK_MATRIX de manière à ce que leurs algorithmes soient exprimés en termes d'opérations portant sur des blocs matrices plutôt que sur de simples éléments scalaires.

Diffusion d'un bloc matrice

À partir de la fonction *local_block*, obtenue par simple renommage de la fonction *item* héritée de la classe ARRAY2[LOCAL_MATRIX] (voir l'exemple 3.9), nous avons mis en œuvre la fonction SPMD correspondante. Cette fonction est reproduite dans l'exemple 3.12.

Dans la définition de la fonction SPMD *block*, on met à profit le fait qu'un objet de type LOCAL_MATRIX est un objet *transmissible*. Au niveau du nœud propriétaire du bloc considéré, la matrice locale retournée par *local_block* et affectée à l'entité **Result** (ligne 6) peut être diffusée avec la simple expression **Result**.*bcast* (ligne 8).

Exemple 3.12

```
block (i, j: INTEGER): like local_block is
    require
        valid_block: dist.valid_block (i, j);
    do
        if dist.block_is_local(i, j) then                                    5
            Result := local_block (i, j);
            -- Broadcast block(i,j) to all nodes
            Result.bcast;
        else
            -- Receive block(i,j) from source                                10
            !!Result.make (dist.limax(i), dist.ljmax(j));
            Result.recv_bcast_from (dist.owner_of_block (i, j));
        end; -- if
    ensure
        local_block: block_is_local(i,j) implies (Result = local_block(i, j));  15
    end; -- block
```

Au niveau de chacun des autres nœuds, on crée une nouvelle matrice locale (ayant la taille appropriée) que l'on affecte à **Result** et qui sert alors de réceptacle pour les données reçues lors de l'exécution de l'expression **Result**.*recv_bcast_from* (lignes 11 et 12).

On notera que, dans cette définition de la fonction *block*, on applique le même mécanisme de rafraîchissement *Refresh* sur la base duquel on a déjà implanté l'accesseur SPMD *item* dans la classe DIST_MATRIX. La différence entre la définition de *block* et celle de *item* réside simplement dans le volume des données transmises lors de l'exécution de l'une ou l'autre routine.

Transfert d'un bloc matrice en point-à-point

La fonction *block* décrite ci-dessus peut être utilisée dans un algorithme parallèle pour obtenir qu'une copie d'un bloc quelconque soit mise à disposition de chaque nœud. Cette fonction ne permet en revanche pas d'obtenir que le rafraîchissement ait lieu sur un nœud particulier. La fonction *bring_block_to* reproduite dans l'exemple 3.13 a été développée dans ce but. Elle ressemble beaucoup à la fonction *block*, mais cette fois des primitives de communication en point-à-point sont utilisées à la place des primitives de diffusion.

Exemple 3.13

```
bring_block_to (i, j, dest: INTEGER): like local_block is
    require
        valid_block: dist.valid_block (i, j);
    do
        if (block_is_local (i, j)) then                                    5
            Result := local_block (i, j);
            if (POM.node_id /= dest) then
                -- Send block(i,j) to dest
                Result.send (dest);
            end; -- if                                                     10
        end; -- if
        if (POM.node_id = dest)
            and (POM.node_id /= owner_of_block (i, j)) then
            -- Receive block(i,j) from source
            !!Result.make (dist.lbimax(i), dist.lbjmax(j));                15
            Result.recv_bcast_from (owner_of_block (i, j));
        end; -- if
    ensure
        local_block: block_is_local(i,j) implies (Result = local_block(i, j));
        valid_result: (POM.node_id = dest) implies Result.deep_equal(block(i,j));20
    end; -- bring_block_to
```

3.4.2 Matrices distribuées par lignes et par colonnes

Nous avons intégré dans la bibliothèque Paladin deux classes supplémentaires, baptisées DROW_MATRIX et DCOL_MATRIX, décrivant des mises en œuvre spécifiques pour les matrices distribuées par lignes et par colonnes respectivement. Ces deux types de distribution correspondent bien sûr à des cas particuliers du schéma général de distribution par blocs, mais nous avons cependant choisi de les doter d'une mise en œuvre *ad hoc*.

Dans les lignes qui suivent, on décrit succinctement la mise en œuvre de la classe DCOL_MATRIX (celle de la classe DROW_MATRIX étant évidemment très semblable).

Représentation interne

La représentation interne des matrices distribuées par colonnes s'appuie, non pas sur une table bi-dimensionnelle de matrices locales comme c'est le cas pour les matrices instanciées d'après la classe DBLOCK_MATRIX, mais sur une table mono-dimensionnelle de vecteurs locaux (instances de la classe LOCAL_VECTOR). La classe DCOL_MATRIX hérite donc à la fois de la classe DIST_MATRIX et de la classe ARRAY[LOCAL_VECTOR] (voir la figure 3.6 et l'exemple 3.14).

Exemple 3.14

```
class DCOL_MATRIX inherit
    DIST_MATRIX
    ARRAY [LOCAL_VECTOR]
        rename
            make as make_table,                                    5
            put as put_column, item as local_column
        end
    ...
invariant
    bfi_valid : (dist.bfi = nrow)                                  10
end -- class DCOL_MATRIX
```

Gestion de la distribution

Nous n'avons pas introduit dans Paladin de classes spécialement dédiées à la gestion des schémas de distribution par lignes ou par colonnes. La classe DISTRIBUTION_2D, bien qu'ayant un champ d'application plus large, peut fort bien assumer cette tâche. On utilise donc, pour décrire la distribution d'une instance de

DCOL_MATRIX un descripteur de distribution de type DISTRIBUTION_2D. Toutefois, l'invariant placé dans le texte de la classe DCOL_MATRIX (exemple 3.14) exprime une contrainte sur le type particulier de distribution pour lequel cette classe propose une mise en œuvre appropriée : il est nécessaire que les paramètres du descripteur de distribution *dist* vérifient la propriété *dist.bfi* = *dist.nrow*, ce qui revient à dire que le partitionnement doit être tel qu'un bloc couvre toute la hauteur d'une matrice distribuée.

Mise en œuvre

Dans la classe DCOL_MATRIX, l'initialisation de la table des vecteurs locaux et la mise en œuvre des accesseurs locaux *local_put* et *local_item* sont réalisées d'une manière très semblable à celle décrite dans le paragraphe 3.4.1 pour la classe DBLOCK_MATRIX.

Nous avons redéfini l'accesseur *column* hérité de la classe MATRIX de manière à ce qu'il retourne, non plus une vue sur la colonne spécifiée, mais directement le vecteur considéré si celui-ci appartient au nœud local, ou sinon une copie locale du vecteur distant (voir l'exemple 3.15).

Exemple 3.15

```
column (j: INTEGER): like local_column is
    do
        if column_is_local (j) then
            Result := local_column (j);
            Result.bcast;                                          5
        else
            !!Result.make (nrow);
            Result.recv_bcast_from (owner_of_column (j));
        end; -- if
    end;                                                           10
```

Dans la nouvelle mise en œuvre de l'accesseur *column*, on s'appuie sur le fait qu'un objet de type LOCAL_VECTOR est un objet *transmissible*. On voit de nouveau apparaître le même mécanisme de rafraîchissement qui a déjà été implanté dans l'accesseur *item* de la classe DIST_MATRIX et dans la fonction *block* de la classe DBLOCK_MATRIX.

Nous avons également doté la classe DCOL_MATRIX d'une fonction permettant la transmission des vecteurs colonnes en mode point-à-point. La mise en œuvre de la fonction *bring_column_to* est très semblable à celle de *bring_block_to* dans la classe DBLOCK_MATRIX.

Discussion

On peut s'interroger sur l'intérêt qu'il y avait à développer les classes DCOL_MATRIX et DROW_MATRIX, alors que la classe DBLOCK_MATRIX peut paraître amplement suffisante puisque autorisant les mêmes schémas de distribution. La raison pour laquelle nous avons malgré tout construit les classes DROW_MATRIX et DCOL_MATRIX est qu'il est en général plus facile d'exprimer — et *a fortiori* de paralléliser — un algorithme dans lequel les opérations élémentaires portent sur des vecteurs plutôt que sur des blocs matrices. En d'autres termes, les classes DROW_MATRIX et DCOL_MATRIX sont plus faciles à optimiser que la classe DBLOCK_MATRIX.

3.4.2.1 Perspectives d'extension

À l'heure actuelle, la bibliothèque Paladin ne permet que la distribution de matrices denses selon les schémas décrits au paragraphe 3.2. De nouvelles classes héritant de DIST_MATRIX et proposant des mises en œuvre alternatives pour les matrices distribuées peuvent toutefois être aisément ajoutées à la bibliothèque.

On pourrait par exemple développer une classe décrivant la mise en œuvre de matrices partitionnées en blocs hétérogènes. En fait, le format de représentation interne choisi pour les matrices distribuées par blocs et décrit au paragraphe 3.4.1 n'oblige nullement à ce que les matrices de ce type soient partitionnées de manière homogène (on peut noter dans la ligne 16 de l'exemple 3.10 que chaque bloc local peut être créé avec une taille différente de celle des autres blocs). En fait, seule la classe DISTRIBUTION_2D impose à l'heure actuelle un partitionnement homogène. Il suffirait donc de construire une classe semblable à DISTRIBUTION_2D mais permettant le partitionnement en blocs hétérogènes pour que les matrices de la classe DBLOCK_MATRIX puissent adopter ce genre de partitionnement.

On pourrait également développer des classes décrivant la mise en œuvre de matrices creuses distribuées, ou encore de matrices triangulaires distribuées, etc.

L'extensibilité de la bibliothèque Paladin est donc bien assurée conformément aux exigences formulées au chapitre 2, puisqu'il est toujours possible d'ajouter à la hiérarchie existante de nouvelles classes décrivant de nouveaux schémas de distribution ou de nouveaux formats de représentation interne pour les matrices et vecteurs.

Chapitre 4

Techniques d'optimisation

4.1 Calculs de localisation des données

4.1.1 Motivation

Accéder à un élément d'un agrégat distribué est une opération coûteuse, notamment en raison des multiples calculs qui sont nécessaires pour localiser cet élément (calcul de l'identité du nœud propriétaire et calcul de l'adresse locale de l'élément sur ce nœud). Lorsqu'on doit implanter les fonctions de localisation des données rendues inévitables par la distribution des agrégats, on est en général amené à faire un choix quant à la technique de mise en œuvre qu'il faut adopter. On peut choisir de réaliser tous les calculs *in extenso* chaque fois que l'on devra identifier le possesseur et l'adresse locale d'un élément donné. Il est aussi parfois possible de réaliser tout ou partie des calculs une fois pour toutes lors de la création d'un agrégat distribué, et de stocker ensuite les résultats de ces calculs dans des tables qu'il suffira de consulter à chaque nouvel accès. Cette deuxième solution est notamment envisageable lorsqu'il faut gérer la distribution de structures de données régulières indiçables telles que des matrices et vecteurs, ou plus généralement des tableaux.

Entre ces deux approches « extrêmes », l'une privilégiant les temps de localisation des données au détriment de l'occupation en mémoire, l'autre économisant la mémoire mais induisant des temps de localisation plus longs, on peut souvent proposer des approches intermédiaires dans lesquelles certains calculs (les plus coûteux) sont effectués une fois pour toutes et les résultats stockés dans des tables, alors que les calculs moins coûteux sont effectués aussi souvent que nécessaire.

4.1.2 Illustration

On a décrit au paragraphe 3.2.2 le rôle et la mise en œuvre de la classe DISTRIBUTION_2D, qui permet de gérer le partitionnement et le placement de structures de données bi-dimensionnelles selon des schémas de distribution inspirés de ceux de HPF.

Dans cette classe, nous avons implanté les fonctions d'identification des nœuds propriétaires des données et de conversion d'indices de la manière la plus simple et la plus directe : les calculs élémentaires évoqués dans les paragraphes 3.2.2 et 3.2.3 sont effectués systématiquement chaque fois qu'on cherche à localiser un bloc ou un élément scalaire particuliers d'une matrice distribuée.

Cette mise en œuvre de la classe DISTRIBUTION_2D est particulièrement économique du point de vue de l'occupation de la mémoire : un descripteur de distribution (instance de la classe DISTRIBUTION_2D) n'occupe pas plus de quelques octets en mémoire. Cependant, les fonctions de la classe DISTRIBUTION_2D étant appelées à être invoquées très fréquemment dès lors qu'on procède à des calculs impliquant une matrice distribuée, nous avons développé une classe descendante de DISTRIBUTION_2D, baptisée FAST_DISTRIBUTION_2D (figure 4.1), dans laquelle certains des calculs sont effectués une fois pour toute dès l'instanciation de la classe, et les résultats stockés dans des tables.

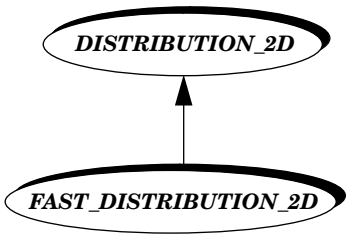


FIG. 4.1 - *La classe* FAST_DISTRIBUTION_2D *est une version optimisée de la classe* DISTRIBUTION_2D

Pour réduire au strict minimum les temps de réponse des fonctions de la classe FAST_DISTRIBUTION_2D, il faudrait calculer et stocker pour chaque couple (i, j) du domaine d'indices global considéré un quintuplet $(owner, bi, bj, li, lj)$ identifiant :

- $owner$: le nœud propriétaire de l'élément (i, j) considéré ;
- (bi, bj) : le bloc de partitionnement englobant cet élément ;
- (li, lj) : les coordonnées locales de l'élément (i, j) dans le bloc (bi, bj).

Pour stocker toutes ces informations, il faudrait créer et maintenir sur chacun des nœuds participant à l'exécution d'une application SPMD des descripteurs de distribution occupant plus de place en mémoire que les matrices elles-mêmes. En effet, pour une matrice de nombres réels en double précision de taille $m \times n$ on devrait créer un descripteur occupant au moins $(20 \times m \times n)$ octets en mémoire[1], alors que la matrice elle-même n'occuperait que $(8 \times m \times n)$ octets dans le pire des cas (c'est-à-dire dans le cas fort improbable d'une matrice « distribuée » sur un seul nœud).

En construisant la classe FAST_DISTRIBUTION_2D, nous avons donc opté pour un compromis entre la réduction des temps de réponse des primitives de cette classe et l'occupation des instances de la classe en mémoire. Nous avons mis en œuvre la classe FAST_DISTRIBUTION_2D de manière à ne stocker que des informations relatives aux nœuds propriétaires des blocs de partitionnement[2]. On construit une table bi-dimensionnelle de taille $bi_{max} \times bj_{max}$ dans laquelle on stocke pour chaque bloc (bi, bj) l'identité du nœud propriétaire de ce bloc.

Cette table est créée et initialisée lors de la création d'un descripteur de distribution de type FAST_DISTRIBUTION_2D. Elle occupe $(4 \times bi_{max} \times bj_{max})$ octets en mémoire sur chaque nœud. Ainsi, pour gérer la distribution d'une matrice de taille 1000×1000 partitionnée en blocs de taille 100×100, on crée sur chaque nœud une table des possesseurs de blocs occupant 400 octets en mémoire, la matrice distribuée occupant quant à elle approximativement $8/P$ Moctets sur chacun des P nœuds participant au calcul. Il faudrait donc distribuer la matrice sur plus de 20000 nœuds (avec seulement 50 éléments scalaires par nœud) pour que le descripteur de distribution occupe plus de place en mémoire que la matrice elle-même. Cet exemple vise simplement à montrer que l'occupation en mémoire de la table des possesseurs de blocs demeure raisonnable. Il faut en outre garder à l'esprit que plusieurs matrices distribuées peuvent partager un même descripteur de distribution (voir § 3.2.4), et par conséquent une même table des possesseurs de blocs.

La table des possesseurs de blocs demeure totalement invisible pour l'utilisateur. Les fonctions *owner_of_block* et *owner_of_item* héritées de la class DISTRIBUTION_2D ont été redéfinies dans la classe FAST_DISTRIBUTION_2D de manière à consulter directement la table plutôt que d'invoquer la fonction de placement fournie par

1. Dans l'environnement Eiffel d'ISE utilisé pour développer Paladin (version 3.2), les objets de type INTEGER et REAL sont représentés sur 32 bits et les objets de type DOUBLE sur 64 bits (pour un processeur de type Sparc).

2. On rappelle que la fonction de placement est choisie librement par l'utilisateur (voir § 3.2.3) et peut donc être d'une complexité quelconque. Il peut donc être intéressant de ne pas devoir recourir un grand nombre de fois à une fonction de placement lorsque celle-ci est très complexe.

l'utilisateur. La classe FAST_DISTRIBUTION_2D présente donc exactement la même interface que son ancêtre : on peut utiliser indifféremment l'une ou l'autre classe pour décrire un schéma de distribution.

Nous avons réalisé quelques séries de mesures visant à déterminer le gain apporté par l'emploi de la table des possesseurs de blocs. Nous avons ainsi mesuré le temps nécessaire pour déterminer l'identité du nœud possesseur d'un bloc (bi, bj) donné lorsque la fonction de placement utilisée est fournie par l'une des classes ROW_WISE_MAPPING ou COLUMN_WISE_MAPPING décrites au paragraphe 3.2.3. Les mesures ont été réalisées sur station de travail de type Sun Sparc 4/75 en considérant la distribution par blocs d'une matrice de taille 1000×1000 partitionnée en blocs de taille 100×100 (d'où une table des blocs et une table des possesseurs de taille 10×10). Ces mesures ont montré que le temps de réponse de la fonction *owner_of_block* est diminué d'environ 35 % lorsqu'on utilise un descripteur de distribution de type FAST_DISTRIBUTION_2D au lieu d'un descripteur de type DISTRIBUTION_2D.

En fait, cette différence est indépendante du nombre de nœuds considérés dans la machine cible et de la taille du domaine d'indices partitionné. En effet, lorsque le descripteur de distribution associé à une matrice est de type DISTRIBUTION_2D, le temps de réponse de la fonction *owner_of_block* comprend :

- le temps nécessaire à l'invocation de la fonction de placement *map_block* sur l'objet de type ROW_WISE_MAPPING ou COLUMN_WISE_MAPPING associé au descripteur de distribution considéré (noter que cette invocation implique la *sélection dynamique* de la fonction *map_block* appropriée) ;

- le temps de réponse de la fonction *map_block*, dont une mise en œuvre possible a été reproduite dans l'exemple 3.3 (page 71).

En revanche, lorsque le descripteur de distribution est un objet de type FAST_DISTRIBUTION_2D, l'invocation de la fonction *owner_of_block* se traduit simplement par la consultation d'une entrée de la table bi-dimensionnelle des possesseurs de blocs encapsulée dans le descripteur de distribution.

4.2 Parallélisation des algorithmes

4.2.1 Motivation

La mise en œuvre d'accesseurs SPMD dans les classes décrivant des agrégats distribués permet de masquer la distribution à l'utilisateur, afin qu'il lui soit possible de manipuler de manière identique un agrégat local ou un agrégat distribué. Ainsi,

la mise en œuvre des accesseurs SPMD *put* et *item* dans la classe DIST_MATRIX suffit pour garantir la transparence de la distribution des matrices du point de vue de l'utilisateur. Si on invoquait par exemple, dans le cadre d'un programme d'application SPMD, la fonction *trace* décrite au paragraphe 2.2.3 sur un objet de type DBLOCK_MATRIX, cette fonction retournerait bien la valeur de la trace de la matrice considérée sur chacun des nœuds participant à l'exécution. L'algorithme séquentiel encapsulé dans la fonction *trace* de la classe MATRIX peut donc bien être considéré comme un algorithme *par défaut* dans la mesure où, n'étant pas dépendant d'un format de représentation interne particulier, il peut être invoqué sur n'importe quel type de matrice, y compris sur une matrice distribuée.

Le simple fait de distribuer un agrégat suffit à affecter légèrement les possibilités offertes à l'utilisateur : ce dernier peut en effet créer et manipuler des agrégats de plus grande taille, puisque la quantité totale de mémoire offerte par une machine parallèle équivaut en général à plusieurs fois celle d'une machine mono-processeur.

Du point de vue des performances observées, en revanche, la simple distribution d'un agrégat permet rarement d'observer des performances satisfaisantes de la part de cet objet, puisque tous les nœuds participant au calcul exécutent la même séquence d'opérations (à l'exception des opérations d'écriture locale). En fait, on observera même souvent une dégradation des performances, chaque accès en lecture aux données élémentaires d'un agrégat distribué impliquant de coûteux échanges de données entre les nœuds de la machine parallèle utilisée. Après avoir mis en œuvre un agrégat distribué de manière satisfaisante, on doit donc s'efforcer d'optimiser en les parallélisant les opérateurs associés.

4.2.2 Techniques de parallélisation utilisées dans Paladin

Plusieurs approches sont envisageables pour procéder à la parallélisation d'algorithmes tels que ceux encapsulés dans la bibliothèque Paladin. Une présentation de ces diverses approches peut être trouvée dans [106].

Dans ce paragraphe sont présentées les techniques simples que nous avons appliquées jusqu'à présent dans Paladin afin de paralléliser les opérateurs des agrégats matrices. Ces techniques sont relativement bien connues : ce sont les mêmes que celles que l'on tente de faire appliquer automatiquement par les compilateurs-parallèliseurs pour langages de type HPF [9, 15, 113]. Le problème se ramène, pour l'essentiel, à appliquer la règle dite « des calculs locaux » et à limiter le coût des échanges de données entre nœuds.

4.2.2.1 Application de la règle des calculs locaux

Principe

La règle des calculs locaux spécifie que chaque nœud manipulant un agrégat distribué ne doit procéder qu'à des calculs affectant les données dont il est propriétaire. Cette règle constitue en fait une extension naturelle de la règle des écritures locales[3] : plutôt que de se contenter d'un schéma d'exécution où tous les nœuds réalisent tous les calculs, mais où seul le nœud propriétaire d'une donnée peut finalement y inscrire le résultat du calcul qui vient d'être effectué (application stricte de la règle des *écritures* locales), on tente d'obtenir un schéma d'exécution où seul le nœud propriétaire de la donnée devant être affectée par le résultat d'un calcul réalise effectivement ce calcul.

Pour paralléliser un opérateur en appliquant la règle des calculs locaux, on doit intervenir dans l'algorithme de cet opérateur. Cette intervention s'effectue le plus souvent là où les calculs sont les plus nombreux : dans le corps des nids de boucles. Pour chaque calcul impliquant un ensemble de données élémentaires $\mathcal{S}$ (source) accédées en lecture, et une donnée élémentaire $\mathcal{C}$ (cible) accédée en écriture (pour y stocker le résultat du calcul), on doit :

1. « Rafraîchir » les données de $\mathcal{S}$, c'est-à-dire en amener une copie temporaire sur le nœud propriétaire de $\mathcal{C}$;
2. « Exécuter » le calcul requis sur le seul nœud propriétaire de $\mathcal{C}$ (en utilisant les copies locales des données de $\mathcal{S}$) et y stocker le résultat du calcul.

Illustration

Considérons un exemple simple : l'addition de matrices distribuées. Un algorithme séquentiel pour ce calcul pourrait être :

pour $i = 1..i_{max}$
  pour $j = 1..j_{max}$
    $A(i,j) := A(i,j) + B(i,j)$
  fpour
fpour

Pour paralléliser cet algorithme, on remplace l'expression située au cœur du nid de boucles par :

- des instructions visant à amener une copie de $B(i,j)$ sur le nœud propriétaire de $A(i,j)$;

3. Dans la plupart des travaux visant au développement de compilateurs-parallélisers de type HPF, la règle des calculs locaux est d'ailleurs très souvent confondue avec la règle des écritures locales.

- une expression de calcul *local* gardée par un test de localité assurant que cette expression ne sera exécutée que sur le nœud propriétaire de $A(i,j)$.

Le cœur du nid de boucles peut ainsi prendre (par exemple) la forme suivante :

si je possède $B(i,j)$ et pas $A(i,j)$ **alors**
    envoyer $B(i,j)$ au propriétaire de $A(i,j)$
fsi
si je possède $A(i,j)$ et pas $B(i,j)$ **alors**
    recevoir $B(i,j)$ du propriétaire de $B(i,j)$
fsi
si je possède $A(i,j)$ **alors**
    calculer localement $A(i,j) := A(i,j) + B(i,j)$
fsi

La technique utilisée ici pour appliquer la règle des calculs locaux s'appuie uniquement sur des commandes d'émission-réception et une expression de calcul local gardées par des tests de localité. Elle n'implique en particulier aucune modification du nid de boucles englobant : on ne modifie ni les bornes, ni le pas des itérations. Pour cette raison, cette technique est communément qualifiée de technique de « résolution à l'exécution » [113]. Il s'agit en fait de la technique de base sur laquelle s'appuient les compilateurs-paralléliseurs de type HPF pour générer du code parallèle (les techniques d'optimisation appliquées par ces compilateurs portent essentiellement sur la vectorisation des communications et sur la restriction des domaines d'itération).

On voit dans l'exemple précédent que la parallélisation nécessite que l'on soit capable de tester l'identité du nœud propriétaire d'un élément quelconque, et de transmettre la valeur de cet élément à un nœud destinataire. Les mécanismes de gestion de la distribution encapsulés dans la classe DISTRIBUTION_2D et les mécanismes de communication offerts par la classe POM répondent à ces besoins et nous permettent donc de paralléliser les opérateurs des matrices distribuées. Ainsi, une traduction possible de l'algorithme précédent en Eiffel est représentée dans l'exemple 4.1.

En fait, l'algorithme de l'exemple 4.1 n'est pas tout à fait équivalent au pseudo-code reproduit plus haut (on utilise un si-alors-sinon dans les lignes 15 à 19), mais le principe de la résolution à l'exécution est néanmoins appliqué.

On notera que cet algorithme parallèle ne pouvant être exécuté que lorsque la matrice courante est une matrice distribuée, nous l'avons encapsulé dans une routine de la classe DIST_MATRIX. En outre, cette routine n'ayant pas exactement le même domaine d'application que la routine par défaut *add* implantée dans la classe MATRIX (la matrice B passée en paramètre doit être ici une matrice distribuée, et

Exemple 4.1

```
class DIST_MATRIX inherit
   MATRIX
feature {DIST_MATRIX} -- Optimized operators
   add_dist (B : DIST_MATRIX) is
      local                                                          5
         i, j : INTEGER ;
         tmp : like item ;
      do
         from i := 1 until i > nrow loop
            from j := 1 until i > ncolumn loop                       10
               if B.item_is_local (i, j) and not item_is_local (i, j) then
                  POM.send (owner_of_item (i, j), B.local_item (i, j)) ;
               end ; -- if
               if item_is_local (i, j) then
                  if not B.item_is_local (i, j) then                 15
                     POM.recv_from (B.owner_of_item (i, j), tmp) ;
                  else
                     tmp := B.local_item (i, j) ;
                  end ; -- if
                  local_put (local_item (i, j) + tmp, i, j) ;        20
               end ; -- if
               j := j + 1 ;
            end ; -- loop
            i := i + 1 ;
         end ; -- loop                                               25
      end ; -- add_dist
   ...
end -- class DIST_MATRIX
```

non plus une matrice quelconque), nous l'avons baptisée *add_dist*. On aura l'occasion de revenir sur le problème de nommage des routines parallèles dans les paragraphes suivants.

Avec la routine *add_dist*, on peut réaliser en parallèle l'addition de deux matrices A et B distribuées. De même que l'opérateur *add* de la classe MATRIX encapsule un algorithme *séquentiel* par défaut capable d'additionner deux matrices *quelconques*, on peut considérer que l'algorithme encapsulé dans la routine *add_dist* est un algorithme *parallèle* par défaut pour l'addition de matrices *distribuées*. On verra plus loin que des algorithmes plus efficaces peuvent être construits, qui sont spécialement adaptés à certains schémas de distribution des opérandes.

Alternatives à la règle des calculs locaux

La règle des calculs locaux n'est pas la seule qui puisse être appliquée. On peut par exemple faire en sorte que tous les nœuds possédant des données impliquées dans une phase de calcul se « concertent » pour « élire » l'un d'entre eux, chargé de procéder à ce calcul et d'en transmettre le résultat au nœud propriétaire de la donnée affectée par ce calcul (cette approche peut se révéler intéressante lorsque les calculs sont irréguliers et que la répartition des données entre les nœuds peut varier dynamiquement). Dans [104], par exemple, les auteurs proposent d'appliquer la règle dite « des calculs *presque* locaux » (*almost owner compute rule*). Avec cette règle la plupart des calculs — mais pas nécessairement tous — sont réalisés par le propriétaire des données devant être affectées par ces calculs. Dans [112], les auteurs proposent de décrire des schémas de partitionnement des calculs, tout comme on décrit déjà le partitionnement des données. Les nœuds effectuant les calculs ne sont alors plus nécessairement ceux qui possèdent les données affectées par les résultats des calculs, mais ceux spécifiés par le schéma de partitionnement des calculs.

4.2.2.2 Réduction du coût des communications

Sur la grande majorité des architectures massivement parallèles actuelles, le coût des communications est moindre lorsqu'on envoie un seul message long à la place de plusieurs messages courts. Ceci est la conséquence du fait que, sur ces machines, la latence des transmissions contribue pour une bonne part au coût total des communications[4] (une étude récente portant sur les communications dans les réseaux de processeurs peut être trouvée dans [107]).

Pour améliorer les performances d'un algorithme parallèle, on a donc tout intérêt

4. Le temps de transfert t d'un message de longueur l sur un médium non chargé obéit en général à une loi de la forme $t = \tau.l + \delta$, où τ désigne la bande passante du médium et δ le temps de latence des communications sur ce médium.

à réduire le coût des communications en supprimant les communications inutiles, et en vectorisant les communications qui ne peuvent être évitées.

Suppression des tests de localité et des communications inutiles

On peut parfois déduire statiquement du schéma de distribution de l'agrégat considéré — ou des agrégats considérés — qu'une donnée est disponible localement sur un nœud. Dans ce cas on peut, au niveau de ce nœud, faire en sorte d'accéder localement à cette donnée en utilisant des accesseurs locaux. Cette technique permet d'économiser les tests de localité et les communications inutiles qui sont autrement impliqués par l'emploi d'accesseurs SPMD. De la même manière, on peut parfois décider statiquement que la valeur d'une donnée distante n'a pas été modifiée entre deux accès en lecture à cette donnée. Dans ce cas, une communication peut être évitée en utilisant une variable temporaire.

Vectorisation des communications

Le terme « vectorisation » doit ici être pris dans son sens le plus large. La véritable communication vectorisée n'est possible que lorsque les données élémentaires devant être transmises sont contiguës en mémoire et peuvent être considérées comme appartenant à une structure de données englobante (un « vecteur »), qu'il est alors possible de transmettre en une seule fois. Dans tous les autres cas, on peut procéder à l'agrégation des données, qui consiste à fabriquer un vecteur temporaire (un objet *tampon*) dans lequel on regroupe des données initialement dispersées dans la mémoire, afin de transmettre ce vecteur en une seule fois. Au niveau du nœud récepteur, on doit procéder à l'opération inverse et extraire du vecteur reçu les données élémentaires qu'il renferme.

Il faut noter qu'il n'est pas toujours judicieux de recourir à tout prix à l'agrégation de données dans un tampon sous prétexte de vectoriser les communications. Avec l'évolution des architectures parallèles, on voit apparaître des machines dans lesquelles la bande passante des canaux de communication est supérieure au débit obtenu lors des transferts de mémoire à mémoire sur un même nœud (c'est par exemple le cas sur la machine Intel Paragon XP/S : nous avons mesuré une bande passante maximale de 82 Mo/s pour les communications en point-à-point[5], contre seulement 66 Mo/s pour les copies de mémoire à mémoire sur un même nœud[6]). Avec de telles machines, il peut dans certains cas être préférable de procéder à plu-

5. Mesures réalisées avec le système OSF/1 et les primitives de communication NX/2.
6. Mesures réalisées en utilisant la primitive `memcpy`.

sieurs émissions successives de petits segments de données plutôt que de chercher à les agréger dans un tampon.

Les mécanismes de communication de la classe POM, l'abstraction des objets transmissibles offerte par la classe TRANSMISSIBLE, et le principe du masquage d'information (grâce au mécanisme de l'encapsulation) permettent d'envisager n'importe quel type de « vectorisation » des communications dans EPEE. Dans Paladin, cependant, les agrégats manipulés étant des vecteurs et matrices denses, nous nous sommes contentés d'appuyer la vectorisation des communications dans les algorithmes parallèles sur les objets transmissibles de la bibliothèque : les vecteurs et matrices locaux.

Illustration

Considérons une fois de plus l'exemple de l'addition de matrices distribuées, et supposons cette fois qu'au lieu de ne rien savoir de la distribution des matrices A et B impliquées dans le calcul, on *sait* que ces matrices sont toutes deux distribuées par colonnes (ce sont donc des instances de la classe DCOL_MATRIX).

Cette fois, l'algorithme parallèle peut être exprimé différemment : on peut tirer parti de la connaissance supplémentaire que l'on a de la distribution des matrices A et B, et faire en sorte que les calculs soient exprimés dans cet algorithme en termes d'opérations vecteur-vecteur, et que les mouvements de données portent sur des vecteurs colonnes et non plus sur de simples éléments scalaires.

pour $j = 1..j_{max}$
  si je possède $B(:,j)$ et pas $A(:,j)$ **alors**
    envoyer $B(:,j)$ au propriétaire de $A(:,j)$
  fsi
  si je possède $A(:,j)$ et pas $B(:,j)$ **alors**
    recevoir $B(:,j)$ du propriétaire de $B(:,j)$
  fsi
  si je possède $A(:,j)$ **alors**
    calculer localement $A(:,j) := A(:,j) + B(i,j)$
  fsi
fpour

En fait, la séquence constituée de l'émission et de la réception d'une colonne de la matrice B peut être exprimée de manière plus concise. En effet, on a mis en œuvre dans ce but précis dans les classes décrivant les matrices distribuées des routines permettant le « rafraîchissement » localisé ou généralisé des blocs de partitionnement. Dans le cas d'une matrice distribuée par colonnes, on dispose ainsi de la fonction *bring_column_to*[7] qui permet d'amener une copie d'un vecteur colonne

7. La fonction *bring_column_to* a été décrite au paragraphe 3.4.1.4.

sur un nœud donné. Cette fonction encapsule exactement les mêmes tests de localité et les mêmes opérations d'émission-réception qui ont été exprimées *in extenso* dans l'algorithme en pseudo-code reproduit plus haut. Cet algorithme peut donc être exprimé de manière plus concise comme illustré ci-dessous, et traduit en Eiffel comme montré dans l'exemple 4.2.

pour $j = 1..j_{max}$
  rafraîchir $B(:,j)$ sur le nœud propriétaire de $A(:,j)$
  si je possède $A(:,j)$ **alors**
    calculer localement $A(:,j) := A(:,j) + B(:,j)$
  fsi
fpour

On notera dans l'exemple 4.2 l'emploi de la fonction *bring_column_to* invoquée sur la matrice B (ligne 10). Cette fonction permet donc de faire l'*abstraction* des détails de mise en œuvre du rafraîchissement d'une colonne de B sur un nœud donné.

Exemple 4.2

```
class DCOL_MATRIX inherit
   DIST_MATRIX
feature {DIST_MATRIX} -- Optimized operators
   add_dcol (B: DCOL_MATRIX) is
      local                                                          5
         j: INTEGER;
         B_col: like local_column;
      do
         from j:= 1 until i > ncolumn loop
            B_col:= B.bring_column_to (owner_of_column (j));         10
            if column_is_local (j) then
               local_column (j).add (B_col);
            end; -- if
            j:= j + 1;
         end; -- loop                                                15
      end; -- add_dcol
   ...
end -- class DCOL_MATRIX
```

On notera également que, l'algorithme parallèle considéré ici ne pouvant être exécuté que pour additionner deux matrices distribuées par colonnes, nous l'avons baptisé *add_dcol* et encapsulé dans la classe DCOL_MATRIX.

4.2.3 Présentation de quelques opérateurs parallèles

Dans ce paragraphe sont présentés quelques uns des opérateurs parallèles que nous avons introduits dans la bibliothèque Paladin. Pour obtenir ces opérateurs parallèles, nous nous sommes contentés d'appliquer la règle des calculs locaux et les techniques de vectorisation des communications évoquées plus haut. Il va de soi qu'il est possible d'encapsuler dans les classes de Paladin des algorithmes parallèles autrement plus complexes — et probablement plus performants — que ceux qui vont être décrits dans les pages qui suivent. Cependant, notre propos est avant tout de montrer qu'il est possible de procéder à une parallélisation « en douceur » de la bibliothèque Paladin, et qu'il n'est pas forcément besoin d'être un expert en algorithmique parallèle pour pouvoir produire en un laps de temps réduit des algorithmes SPMD aux performances satisfaisantes.

Exemple 1 : calcul de la trace d'une matrice

Considérons l'opérateur fonction *trace*, tel qu'il a été défini dans la classe MATRIX. L'algorithme associé à cet opérateur dans la classe MATRIX est purement séquentiel et ne tient donc absolument pas compte de la distribution possible de la matrice pour laquelle on désire calculer la somme des éléments diagonaux. Nous avons redéfini la fonction *trace* dans la classe DIST_MATRIX, en la dotant d'un algorithme de calcul mieux adapté aux caractéristiques des matrices distribuées. Le calcul de la trace se ramène alors à une opération classique de réduction : chaque nœud calcule d'abord localement une trace partielle en ne considérant que les éléments scalaires dont il est propriétaire, puis il diffuse le résultat à l'intention des autres nœuds. Le calcul se termine sur chaque nœud lorsque toutes les traces partielles ont été reçues et additionnées. La nouvelle définition de l'opérateur fonction *trace* est reproduite dans l'exemple 4.3.

Le calcul de la trace s'exprime à présent comme une réduction SPMD, réalisée en invoquant la routine de réduction *reduce* d'un objet agent [8] de type DIST_REDUCTOR (ligne 23). La classe DIST_REDUCTOR, décrivant les agents capables de procéder à des réductions dans un contexte SPMD, est décrite en détails dans l'annexe B. C'est l'un des « outils » de parallélisation intégrés à l'environnement EPEE.

Chaque nœud doit tout d'abord calculer une trace partielle en ne considérant que les éléments diagonaux dont il est propriétaire. Dans la fonction *local_trace*, on applique la règle des calculs locaux en effectuant un test de localité, réalisé en invoquant la fonction *item_is_local* sur le descripteur de distribution *dist* de la

8. La notion d'objet agent a été introduite au paragraphe 3.2.4. Les agents sont des abstractions algorithmiques, c'est-à-dire des objets capables d'agir sur d'autres objets.

Exemple 4.3

```
class DIST_MATRIX
inherit
    MATRIX
        redefine trace end
feature {NONE} -- Private operators                                    5
    local_trace: like item is
            -- Compute the sum of local diagonal items
        local
            i: INTEGER;
        do                                                             10
            from i := 0 until i > nrow loop
                if dist.item_is_local (i, i) then
                    Result := Result + local_item (i, i);
                end; -- if
            end; -- loop                                               15
        end; -- local_trace
feature -- Optimized operators
    trace: like item is
        local
            action: SUM [like item];                                   20
            dist_reductor: DIST_REDUCTOR [like item];
        do
            Result := dist_reductor.reduce (local_trace, action);
        end; -- trace
    ...                                                                25
end -- class DIST_MATRIX
```

matrice courante (ligne 12). Le résultat retourné par la fonction *local_trace* est passé en paramètre lors de l'invocation de l'opérateur de réduction (ligne 23). Le second paramètre décrit l'opération binaire — ici une simple somme arithmétique — devant être utilisée pour réduire l'ensemble des traces partielles et évaluer la trace réelle de la matrice courante.

Exemple 2 : somme de matrices distribuées par blocs

Considérons à présent le cas de l'opérateur d'addition *add*, qui prend en paramètre une matrice et l'ajoute à la matrice courante. La mise en œuvre de cette procédure dans la classe MATRIX est telle que l'on peut additionner deux matrices quelconques référencées par deux entités A et B en introduisant dans un programme SPMD l'expression *A.add(B)*. Cependant cette mise en œuvre est purement séquentielle et, par conséquent, particulièrement inefficace dès lors que l'une au moins des deux matrices A et B est une matrice distribuée. On a déjà détaillé au paragraphe 4.2.2 la mise en œuvre de deux versions parallèles de l'opérateur d'addition, la première dédiée aux matrices distribuées en général (c'est-à-dire capable d'additionner deux matrices distribuées, quel que soit leur schéma de distribution), et la seconde spécialement adaptée aux matrices distribuées par colonnes (il va de soi qu'une mise en œuvre parallèle adaptée aux matrices distribuées par lignes peut être réalisée de la même manière).

En développant la classe DBLOCK_MATRIX, nous y avons inséré un algorithme capable d'additionner efficacement deux matrices distribuées par blocs, pourvu que ces deux matrices aient le même partitionnement. Cet algorithme, dont le code est reproduit dans l'exemple 4.4, pourrait être exprimé ainsi en pseudo-code[9] :

```
pour i = 0..bi_max
    pour j = 0..bj_max
        rafraîchir B_ij sur le nœud propriétaire de A_ij
        si je possède A_ij alors
            calculer localement A_ij := A_ij + B_ij
        fsi
    fpour
fpour
```

On notera dans l'exemple 4.4 que :

- la précondition de la ligne 5 fixe le domaine d'application de l'opérateur *add_dblock* : les deux matrices impliquées dans le calcul — *i.e.* la matrice

9. Dans ce document, on note toujours A_{ij} un bloc de la matrice A, et $A(i,j)$ un élément scalaire de la même matrice.

Exemple 4.4

```
add_dblock (B: DBLOCK_MATRIX) is
    require
        B_valid: (B /= Void);
        same_size: (nrow = B.nrow) and (ncolumn = B.ncolumn);
        same_dist: (dist.bfi = B.dist.bfi) and (dist.bfj = B.dist.bfj);
    local
        bi, bj: INTEGER;
        Bs_block: LOCAL_MATRIX;
    do
        from bi := 0 until bi > dist.nbimax loop
            from bj := 0 until bj > dist.nbjmax loop
                -- Refresh B(bi,bj) on the owner of C(bi,bj)
                Bs_block := B.bring_block_to (owner_of_block (bi, bj));
                if (block_is_local (bi, bj)) then
                    -- Add B(bi,bj) to C(bi,bj) locally
                    local_block (bi, bj).add (Bs_block);
                end; -- if
                bj := bj + 1;
            end; -- loop
            bi := bi + 1
        end; -- loop
    end; -- add_dblock
```

« courante » et la matrice B passée en paramètre — doivent être partitionnées en blocs de même taille. Ceci n'implique cependant pas que la fonction de placement des deux matrices doive être identique (auquel cas plus aucun échange de données ne serait nécessaire entre les nœuds participant au calcul).

- On utilise la fonction *bring_block_to*[10] (ligne 13) pour « rafraîchir » le bloc B_{ij} sur le nœud propriétaire du bloc A_{ij} (A étant la matrice courante).

- On réalise une commande gardée par un test de localité de telle manière que chaque nœud ne procède plus qu'à des calculs locaux (ligne 14).

Vectorisation et accès à la mémoire

Au lieu d'être exprimé à l'aide des accesseurs de base *put* et *item*, l'algorithme de *add_dblock* est directement exprimé en termes d'opérations portant sur des blocs matrices. Les communications se trouvent ainsi vectorisées naturellement, du fait que l'on manipule directement des blocs matrices et non plus de simples valeurs scalaires.

En fait, vectoriser les accès aux données n'a pas pour seule conséquence d'améliorer les performances des communications. Le fait de restructurer les algorithmes de la classe MATRIX et des classes descendantes de manière à ce qu'ils procèdent à des opérations portant sur les blocs matrices aide à améliorer les performances globales de Paladin, parce que cela contribue à réduire le coût de la pagination — voire celui des défauts de cache — sur la plupart des processeurs modernes. Utiliser des accesseurs de haut niveau pour manipuler les agrégats matrices entraîne donc une amélioration des performances des opérateurs, non seulement grâce à une réduction du coût des échanges de données, mais aussi grâce à une meilleure exploitation de l'architecture de la mémoire.

Cette technique a déjà été mise en pratique dans d'autres bibliothèques : la bibliothèque LAPACK [7] améliore les performances des bibliothèques LINPACK et EISPACK en restructurant les algorithmes de manière à ce qu'ils procèdent à des opérations portant sur des blocs matrices à l'intérieur des nids de boucles, et en invoquant des routines de type BLAS 3 [88] partout où c'est possible. Dans le noyau BLAS, les opérations portant sur les blocs matrices sont optimisées pour exploiter au mieux les caractéristiques architecturales des machines à mémoire partagée. La bibliothèque ScaLAPACK [44] est construite selon la même idée, mais l'utilisation d'opérations portant sur les blocs matrices vise cette fois à réduire le coût des transmissions entre les nœuds d'une machine à mémoire distribuée.

10. La routine *bring_block_to* a été présentée en détails au paragraphe 3.4.1.4.

Remarque sur le nommage des opérateurs

Il importe de bien comprendre pourquoi nous avons introduit dans DBLOCK_MATRIX un nouvel opérateur parallèle *add_dblock* au lieu de simplement redéfinir dans cette classe l'opérateur *add* hérité de MATRIX.

La signature donnée à l'opérateur *add* dans la classe MATRIX indique que cet opérateur doit être capable de calculer la somme de deux matrices quelconques, qu'elles aient ou non le même format de représentation interne. L'algorithme encapsulé dans *add_dblock*, quant à lui, ne peut calculer que la somme de deux matrices distribuées par blocs. Cet algorithme ne peut donc convenir pour mettre en œuvre l'opérateur *add* dans la classe DBLOCK_MATRIX, car bien qu'il n'y ait dans cette classe aucune ambiguïté quant au type de la matrice courante, la matrice B passée en paramètre à l'opérateur *add* peut elle être une matrice quelconque.

Il faut noter que les règles du typage en Eiffel *permettraient* de redéfinir l'opérateur d'addition dans la classe DBLOCK_MATRIX en lui donnant la signature *add(B : DBLOCK_Matrix)*, ce qui aurait pour conséquence d'imposer à l'entité B de référencer une matrice distribuée par blocs. Cependant, cette redéfinition consituerait une violation de l'objectif d'interopérabilité que nous nous sommes fixés dès le début du développement de Paladin. En effet, alors que la signature de l'opérateur *add* dans la classe abstraite MATRIX indique que l'on doit pouvoir ajouter deux matrices quelconques, altérer la signature de cet opérateur dans la classe DBLOCK_MATRIX aurait pour conséquence de restreindre le champ des opérations possibles : on ne pourrait plus ajouter n'importe quelle matrice à une matrice distribuée par blocs, mais seulement une autre matrice distribuée par blocs (et qui plus est ayant le même partitionnement).

Le problème n'est donc pas d'ordre syntaxique, mais bien d'ordre sémantique : parce que les services offerts par les opérateurs *add* et *add_dblock* ne sont pas exactement équivalents, ces opérateurs doivent tous deux être maintenus simultanément dans DBLOCK_MATRIX.

Ceci n'implique pourtant pas qu'un utilisateur doive spécifier explicitement lequel de ces deux opérateurs est le mieux adapté pour calculer la somme de deux matrices. Cette approche ne saurait satisfaire l'objectif de maintien d'une interface homogène énoncé au paragraphe 1.3.3. On verra dans le paragraphe 4.4 comment nous avons obtenu la sélection automatique de l'opérateur approprié dans la bibliothèque Paladin.

Exemple 3 : factorisation de Gram-Schmidt

L'algorithme séquentiel de décomposition de Gram-Schmidt a été présenté au chapitre 2. Dans cet algorithme la plupart des calculs élémentaires sont exprimés en termes d'opérations portant sur des vecteurs colonnes. En conséquence, il se prête

particulièrement bien à la parallélisation lorsque les matrices impliquées dans le calcul sont distribuées par colonnes. Nous avons développé un nouvel algorithme capable de décomposer une matrice A en un produit $Q.R$ (Q recouvrant A) de manière concurrente, pourvu que A et R soient toutes deux de type DCOL_MATRIX et aient le même schéma de distribution. Cet algorithme n'étant utilisable que pour décomposer une matrice distribuée par colonnes, il a été encapsulé dans la classe DCOL_MATRIX. Étant en outre plus restrictif que son homologue séquentiel de la classe MATRIX en ce qui concerne la matrice R (qui doit impérativement être de type DCOL_MATRIX et non plus simplement de type MATRIX), nous avons dû l'encapsuler dans une nouvelle routine baptisée *mgs_dcol*[11].

L'algorithme parallèle de décomposition de Gram-Schmidt est reproduit ci-dessous en pseudo-code.

pour $k = 1..n$
  si je possède $A(:,k)$ **alors**
    $R(k,k) := \|A(:,k)\|_2$ *(1)*
    $A(:,k) := A(:,k)/R(k,k)$ *(2)*
  fsi
  amener une copie de $A(:,k)$ sur chaque nœud
  pour $j = k+1..n$
    si je possède $A(:,j)$ **alors**
      $R(k,j) := A(:,k)^T A(:,j)$ *(3)*
      $A(:,j) := A(:,j) - A(:,k).R(k,j)$ *(4)*
  fsi
  fpour
fpour

Lors de l'exécution de la routine *mgs_dcol*, les phases de calcul (1) et (2) sont réalisées localement par le seul possesseur du vecteur colonne k (les matrices A et R devant être distribuées à l'identique, le nœud propriétaire de la colonne k de A est également propriétaire de la colonne k de R). Une copie du vecteur k de A est ensuite amenée sur tous les nœuds afin que chacun d'entre eux puisse procéder à la mise à jour des colonnes j dont il est propriétaire en exécutant localement les phases de calcul (3) et (4).

Le code de la routine *mgs_dcol* est reproduit dans l'exemple 4.5.

On notera que :

- seuls des accesseurs locaux (en l'occurrence *local_put*, *local_item* et *local_column*) sont utilisés dans l'algorithme, afin d'éviter les communications et tests de localité inutiles impliqués par l'emploi des accesseurs SPMD ;

11. Le suffixe *_dcol* faisant référence au type requis pour la matrice passée en paramètre.

- les tests de localité effectués à l'aide de la routine *column_is_local* (lignes 17 et 27) permettent de réduire le domaine d'itération sur chaque nœud ;

- les communications ont lieu lors de l'exécution de la fonction *column*[12]. Elles sont naturellement vectorisées du fait que les données manipulées sont des vecteurs locaux (on rappelle que la classe DCOL_MATRIX est mise en œuvre sous la forme d'une table de vecteurs locaux et qu'un objet de type LOCAL_VECTOR est un objet transmissible).

Discussion

Le fait que nous ayons choisi de développer dans Paladin un algorithme parallèle de décomposition de Gram-Schmidt adapté aux schémas de distribution par colonnes résulte de l'observation de l'algorithme séquentiel correspondant. Celui-ci étant exprimé en termes d'opérations portant sur des vecteurs colonnes, il était « logique » de développer – au moins dans un premier temps — un algorithme parallèle adapté aux matrices distribuées par colonnes. Ce choix ne doit pourtant pas être interprété comme signifiant qu'il est impossible d'introduire dans Paladin des algorithmes de décomposition de Gram-Schmidt adaptés à d'autres types de distribution. Ainsi, on pourrait très certainement développer un algorithme adapté aux matrices distribuées par blocs, mais il faudrait toutefois exprimer dans cet algorithme tous les calculs en termes d'opérations portant sur des blocs matrices. On pourrait également envisager le cas où les matrices impliquées dans la décomposition $A \rightarrow Q.R$ ont un schéma de distribution différent, voire même un type différent.

L'extensibilité de la bibliothèque Paladin permet de l'enrichir à tout instant en y intégrant de nouveaux algorithmes satisfaisant les exigences particulières d'un utilisateur, que ces exigences portent sur les schémas de distribution des opérandes ou sur d'autres critères. L'intégration dans la bibliothèque de plusieurs routines, toutes capables de réaliser le même calcul mais dans des conditions différentes, ne doit pas constituer un problème pour l'utilisateur de la bibliothèque. On montrera au paragraphe 4.4 comment on peut faire en sorte que la routine encapsulant l'algorithme le plus approprié pour traiter un problème donné soit sélectionnée dynamiquement et de manière transparente pour l'utilisateur.

Exemple 4 : produit de matrices

De très nombreux algorithmes ont été développés pour calculer le produit de matrices sur machine parallèle. Ces algorithmes s'appuient souvent sur des mouve-

12. La fonction *column* a été décrite au paragraphe 3.4.2. Elle procède à la diffusion du vecteur local désigné afin que chaque nœud puisse ensuite disposer d'une copie locale de ce vecteur.

Exemple 4.5

```
class DCOL_MATRIX inherit
   ...
feature
   mgs_dcol (R: DCOL_MATRIX) is
         -- Modified Gram-Schmidt (Q.R decomposition)                    5
         -- Q.R <- Current (Q overwrites Current)
      require
         rank_ok: (Current.rank = ncolumn);
         size_ok: (nrow = R.nrow) and (nrow = R.ncolumn);
         dist_ok: (dist.bfj = R.dist.bfj);                              10
      local
         k, j: INTEGER;
         col_k: like local_column;
      do
         from k:= 1 until k > ncolumn loop                              15

            if column_is_local (k) then
               -- Update column k locally
               R.local_put (local_column(k).nrm2, k, k);  -- (1)
               local_column (k).scal (1.0 / R.local_item (k, k));  -- (2)  20
            end; -- if

            -- Refresh column k
            col_k:= column (k);

                                                                        25
            from j:= k+1 until j > ncolumn loop
               if column_is_local (j) then
                  -- Update column j locally
                  R.local_put (col_k.dot (local_column (j)), k, j); -- (3)
                  local_column (j).axpy (-R.local_item (k, k), col_k);  -- (4)  30
               end; -- if

               j:= j + 1;
            end; -- loop
            k:= k + 1;                                                  35
         end; -- loop
      end; -- mgs_dcol
   ...
end -- DCOL_MATRIX
```

ments de données complexes, comme par exemple des décalages circulaires ou des permutations des lignes, des colonnes, ou des blocs des matrices distribuées impliquées dans le calcul [27, 41]. En outre, ces algorithmes sont souvent mis en œuvre de manière à offrir des performances optimales sur une architecture donnée [73], voire sur une machine donnée.

Pour paralléliser le calcul du produit de matrices dans Paladin, nous nous sommes jusqu'à présent contentés d'appliquer la simple règle des calculs locaux. Nous avons intégré dans les diverses classes décrivant les matrices distribuées plusieurs algorithmes parallèles, chacun de ces algorithmes étant spécialement adapté pour traiter un schéma de distribution particulier des trois matrices impliquées dans le calcul (les deux matrices opérandes A et B et la matrice résultat C). Parmi ces divers algorithmes, celui de la routine *mult_dblock_dblock* définie dans la classe DBLOCK_MATRIX est spécialement dédié au calcul du produit de matrices lorsque les trois matrices A, B et C sont de type DBLOCK_MATRIX et vérifient en outre les propriétés suivantes :

- les blocs de partitionnement de A et de C ont la même hauteur ;

- les blocs de A couvrent toute la largeur de la matrice ;

- les blocs de partitionnement de C et de B ont la même largeur ;

- les blocs de B couvrent toute la hauteur de la matrice.

Ces contraintes sont exprimées sous forme de préconditions dans l'entête de l'opérateur *mult_dblock_dblock*, reproduit dans l'exemple 4.6 à la page 114. La figure 4.2 illustre le type de distribution admise par cet opérateur.

Dans le corps de l' opérateur *mult_dblock_dblock*, nous avons exprimé l'algorithme de calcul du produit de matrices en termes de produits élémentaires de blocs matrices. Cet algorithme est reproduit ci-dessous en pseudo-code.

pour $i = 0..bi_{max}$
    amener une copie de A_{i0} sur tous les nœuds
    pour $j = 0..bj_{max}$
        amener une copie de B_{0j} sur le nœud propriétaire de C_{ij}
        si je possède C_{ij} **alors**
            calculer $C_{ij} := A_{i0} \times B_{0j}$
        fsi
    fpour
fpour

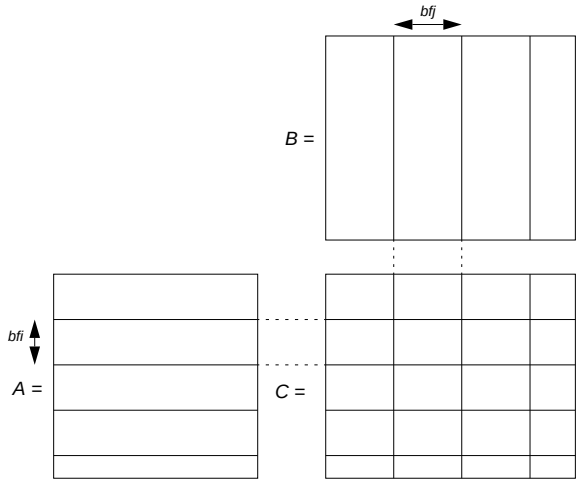


FIG. 4.2 - *Type de distribution admis par l'opérateur* mult_dblock_dblock

Cet algorithme, dont la traduction en Eiffel est reproduite dans l'exemple 4.6, est organisé autour de deux boucles imbriquées. Dans la boucle externe indicée par la variable i, une copie du bloc A_{i0} est mise à la disposition de chaque nœud. Cette opération peut être réalisée en invoquant la fonction *block*[13] de la classe DBLOCK_MATRIX. Dans la boucle interne indicée par la variable j, une copie du bloc B_{0j} est mise à la disposition du propriétaire de C_{ij}, qui calcule alors la nouvelle valeur de ce bloc en procédant au produit de blocs matrices $C_{ij} = A_{i0} \times B_{0j}$. La copie du bloc B_{0j} peut être amenée sur le possesseur de C_{ij} en invoquant la fonction *bring_block_to* de la classe DBLOCK_MATRIX[14].

On peut évaluer grossièrement le coût de l'algorithme parallèle de l'opérateur *mult_dblock_dblock* en termes de communications.

- Chaque bloc de la matrice A est diffusé une fois et une seule lors de l'exécution de la fonction *block*.

- Chaque bloc de la matrice B est transmis au plus $bi_{max}+1$ fois en mode point-à-point : le nombre de blocs effectivement émis et reçus lors de l'exécution de *bring_block_to* dépend du placement relatif des blocs de B et des blocs de C.

Il est possible de jouer sur la fonction de placement et sur la taille des blocs de la matrice C afin d'obtenir que tous les blocs occupant une même colonne dans la table des blocs aient le même propriétaire. Ainsi, si on décrit la distribution d'une

13. La fonction *block* a été présentée au § 3.4.1.4. Elle procède à la diffusion du bloc désigné afin que chaque nœud dispose ensuite d'une copie locale de ce bloc.

14. La fonction *bring_block_to* a été introduite au § 3.4.1.4.

Exemple 4.6

```
class DBLOCK_MATRIX inherit
   ...
feature -- Optimized operators
   mult_dblock_dblock (A, B : DBLOCK_MATRIX) is
         -- C <- A * B, with C = Current.                              5
      require
         size_ok : (nrow = A.nrow) and (ncolumn = B.ncolumn)
                   and (A.ncolumn = B.nrow);
         A_valid : A.dist.bfj = A.ncolumn;
         B_valid : B.dist.bfi = B.nrow;                               10
         C_valid : (dist.bfi = A.dist.bfi) and (dist.bfj = B.dist.bfj);
      local
         i, j : INTEGER;
         A_i, B_j : like local_block;
      do                                                              15
         from i := 0 until i > A.dist.nbimax loop
            -- Refresh A(i,:)
            A_i := A.block (i, 0);
            from j := 0 until j > B.dist.nbjmax loop
               -- Refresh B(:,j) on the owner of C(i,j)              20
               B_j := B.bring_block_to (0, j, owner_of_block (i, j));
               if (block_is_local (i, j)) then
                  -- Compute C(i,j) = A(i,:) * B(:,j)
                  local_block (i, j).mult (A_i, B_j);
               end; -- if                                             25
               j := j + 1;
            end; -- loop j
            i := i + 1;
         end; -- loop i
      end; -- mult_dblock_dblock                                      30
   ...
end -- DBLOCK_MATRIX
```

matrice C de taille $m \times n$ en lui donnant la fonction de placement de la classe ROW_WISE_MAPPING[15] et que P désigne le nombre de nœuds dans la machine cible, il suffit par exemple de donner au facteur de partitionnement *bfj* la valeur $\lceil n/P \rceil$ pour que tous les blocs occupant une même colonne dans la table des blocs de C aient le même propriétaire. Plus généralement[16], cette propriété sera vérifiée lorsqu'on aura $\lceil n/bfj \rceil \mathbf{mod}\ P = 0$.

En distribuant la matrice C de manière à ce que cette propriété soit vérifiée et en donnant aux matrices A et B des distributions « compatibles » avec celle de C (c'est-à-dire respectant les contraintes exprimées dans les préconditions de la routine *mult_dblock_dblock*), l'exécution de cette routine ne nécessite plus aucune transmission des blocs de B car il y a alors identité des nœuds propriétaires de B_{0j} et de C_{ij}, quels que soient i et j. Cette configuration permet d'obtenir des performances optimales pour l'algorithme encapsulé dans *mult_dblock_dblock*. On dira que lorsque les trois matrices A, B et C sont distribuées conformément au schéma décrit ci-dessus, la distribution est *idéale* (du point de vue des communications) pour l'algorithme de l'opérateur *mult_dblock_dblock*.

4.2.4 Observer pour mieux optimiser

Il n'est pas toujours aisé de trouver la distribution idéale des matrices et vecteurs manipulés dans un algorithme parallèle ni, inversement, de mettre en œuvre l'algorithme parallèle capable d'exploiter au mieux certaines caractéristiques de distribution des opérandes. Le comportement d'un algorithme parallèle à l'exécution dépend en outre d'un grand nombre de facteurs, parmi lesquels on peut citer les dimensions des matrices et vecteurs manipulés, leurs schémas de distribution respectifs, le nombre de nœuds impliqués dans le calcul, les caractéristiques physiques des supports de communication dans la machine cible (latence, bande passante, etc.), la topologie du réseau de communication de cette machine (bus Ethernet pour un réseau de stations de travail ; grille, tore ou hypercube pour un super-calculateur), etc.

Les mécanismes d'observation implantés dans la bibliothèque POM peuvent aider à l'expérimentation et à la mise au point des algorithmes parallèles tels que ceux que nous avons implantés dans les classes de Paladin.

15. Les mécanismes de placement de Paladin ont été décrits dans le § 3.2.3.

16. On n'est pas toujours en mesure de choisir librement les facteurs de partitionnement d'une matrice.

Étude des dépendances causales

Les mécanismes d'estampillage vectoriel intégrés à la bibliothèque POM peuvent aider à visualiser après l'exécution d'un algorithme parallèle le graphe des dépendances causales des événements observés dans cet algorithme.

Pour illustrer cette idée, nous avons exécuté sur la machine Paragon XP/S de l'IRISA l'algorithme de calcul du produit de matrices encapsulé dans l'opérateur *mult_dblock_dblock* présenté au paragraphe précédent.

Nous nous sommes tout d'abord placés dans un contexte de distribution « idéale », pour calculer un produit de matrices de la forme $C = A \times B$ sur 4 nœuds de la Paragon. Dans cette première expérience, les matrices B et C étaient de taille 450×550, et la matrice A était de taille 450×450. Nous avons donné au facteur de partitionnement *bfj* la valeur 150 (ce qui vérifie bien la propriété $\lceil 550/150 \rceil \mathbf{mod}\ 4 = 0$ énoncée au paragraphe précédent), et fixé librement à 100 la valeur du facteur *bfi*.

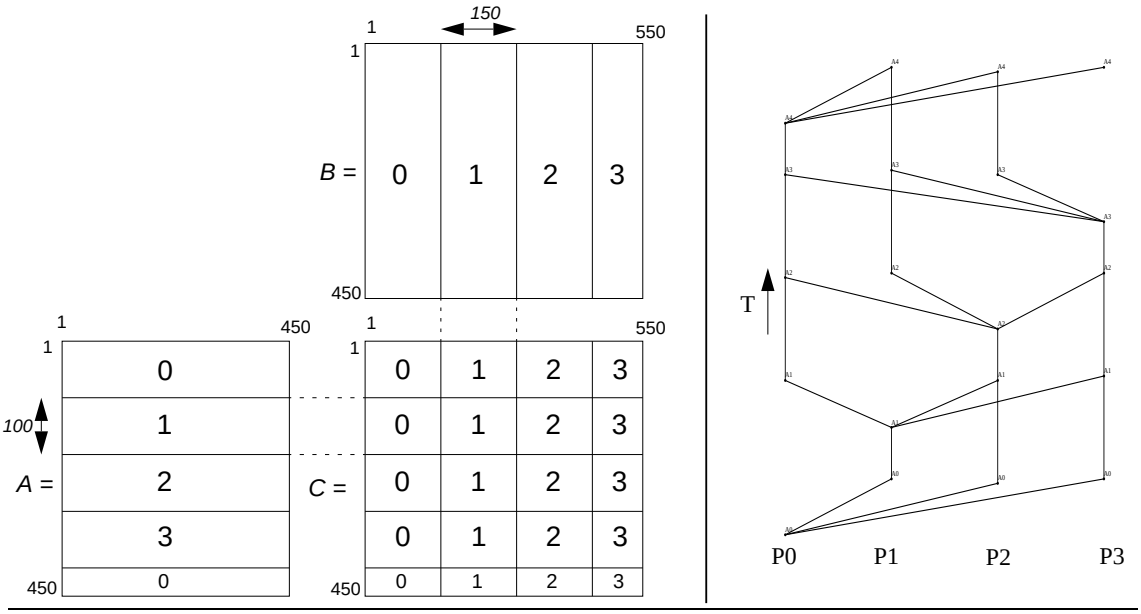


FIG. 4.3 - *Exemple de distribution « idéale » pour l'opérateur* mult_dblock_dblock *(à gauche), et visualisation des échanges de données pendant l'exécution de cet opérateur (à droite)*

On a représenté dans la figure 4.3 (en partie gauche) la distribution ainsi obtenue pour les trois matrices A, B et C. Chaque bloc porte le numéro du nœud qui en est propriétaire.

La routine *mult_dblock_dblock* a été exécutée sur 4 nœuds de la machine Paragon.

Au cours de cette exécution, les services de génération automatique de trace de la POM ont été activés, un message de trace portant une estampille vectorielle étant généré lors de chaque émission ou réception d'un bloc pendant la redistribution. Les messages de trace ont été collectés et traités par un programme observateur fonctionnant en parallèle avec l'application. Les informations produites par l'observateur ont ensuite été fournies à un outil de visualisation graphique capable de dessiner le diagramme des dépendances causales (en fait, un diagramme de Hasse) à partir d'une liste d'événements estampillés.

On a reproduit dans la figure 4.3 (en partie droite) le diagramme des dépendances causales obtenu lors de l'exécution. On peut constater sur ce diagramme que les seules communications ayant lieu au cours de l'exécution sont des diffusions. Il s'agit en fait des diffusions successives des cinq blocs de la matrice A.

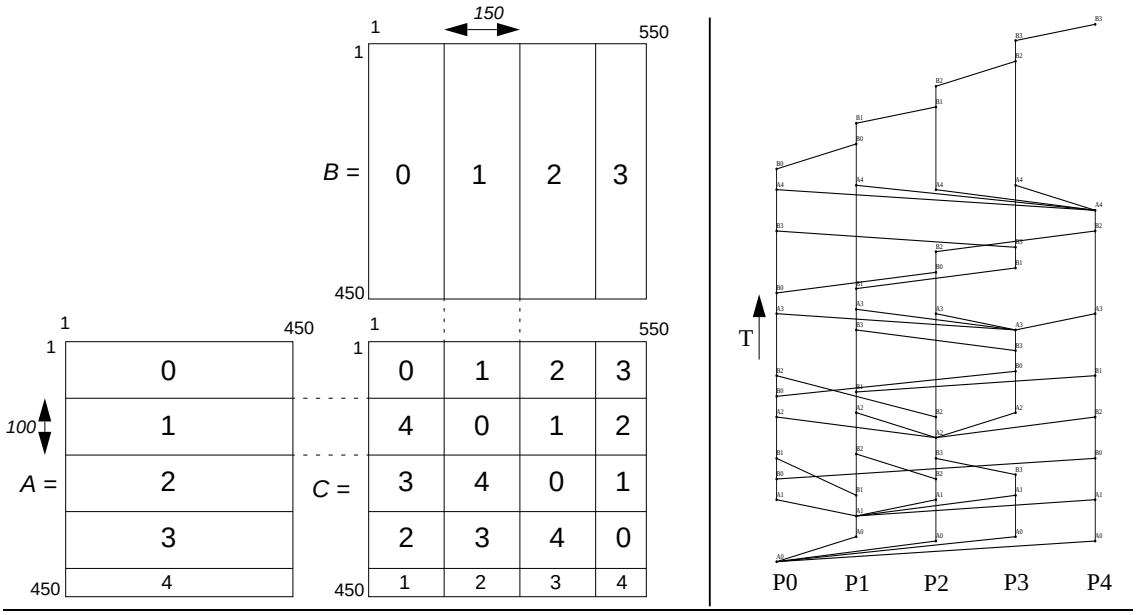


FIG. 4.4 - *Exemple de distribution non idéale pour l'opérateur* mult_dblock_dblock *(à gauche), et visualisation des échanges de données pendant l'exécution de cet opérateur (à droite)*

Nous avons ensuite renouvelé l'expérience du produit de matrices, en utilisant cette fois 5 nœuds de la machine Paragon au lieu de 4. La distribution des matrices A, B et C, et le diagramme des dépendances causales obtenu à l'exécution ont été reproduits dans la figure 4.4.

Pour cette expérience, on ne se trouve plus dans le cadre d'une distribution « idéale » des trois matrices opérandes, comme le confirme d'ailleurs la relation

$\lceil 550/150 \rceil \mathbf{mod}\ 5 \neq 0$. Dans le diagramme des dépendances causales, on voit de nouveau apparaître les diffusions des cinq blocs de la matrice A, mais il s'y ajoute de nombreuses transmissions en point-à-point, qui correspondent aux transmissions des blocs de la matrice B lorsque le nœud propriétaire d'un bloc C_{ij} donné n'est pas en même temps propriétaire du bloc B_{0j}.

Étude du comportement temporel

La génération du graphe de dépendances causales apporte des informations concernant les inter-dépendances entre les événements au cours d'une exécution répartie. En examinant ce graphe, on peut par exemple identifier des contraintes de synchronisation trop fortes dans certains algorithmes parallèles. Cependant le graphe des dépendances causales nous informe sur le parallélisme *potentiel* permis par un algorithme parallèle. Le mécanisme de temps global intégré à la bibliothèque POM permet de dater les événements, et d'examiner ainsi le comportement temporel effectif d'une exécution répartie.

On a reproduit dans la figure 4.5 les diagrammes temporels obtenus à la suite des expériences évoquées précédemment. Ces diagrammes nous apportent une information complémentaire de celle fournie par le diagramme des dépendances causales.

On peut par exemple comparer les durées respectives des phases de calcul et des phases de communication au cours de l'exécution. Ainsi, on constate dans le diagramme de gauche de la figure 4.5 que la première phase de calcul réalisée sur le processeur P1 (calcul de C_{01} en fonction de A_{00} et de B_{01}), qui correspond dans le diagramme au segment borné par les événements E1 et E2, a duré environ 0.25 seconde, alors que le bloc matrice A_{10} diffusé ensuite par P1 a été reçu par le nœud P0 (événement E3) seulement 0.02 secondes plus tard. Ceci confirme que, sur la machine Paragon, les échanges de données sont *très* rapides (on n'observerait pas du tout le même comportement temporel si on réalisait la même expérience sur un réseau de stations de travail reliées par un câble Ethernet, alors que le diagramme des dépendances causales serait identique).

Les deux diagrammes de la figure 4.5 nous permettent aussi de constater que la première phase de calcul dure sensiblement plus longtemps que les phases de calcul suivantes. Ce phénomène est probablement dû au fait que, sur la machine Paragon, les accès à la mémoire cache sont très coûteux. Or, dans notre expérience les matrices A, B et C sont chargées en mémoire lors de la première phase de calcul et y demeurent ensuite pour les phases suivantes.

On peut encore constater, dans le diagramme de droite de la figure 4.5 que certains échanges de données en point-à-point semblent durer beaucoup plus longtemps que d'autres. En fait, le diagramme temporel traduit le fait qu'au cours de l'exécution la réception de certains messages est différée sur certains nœuds parce

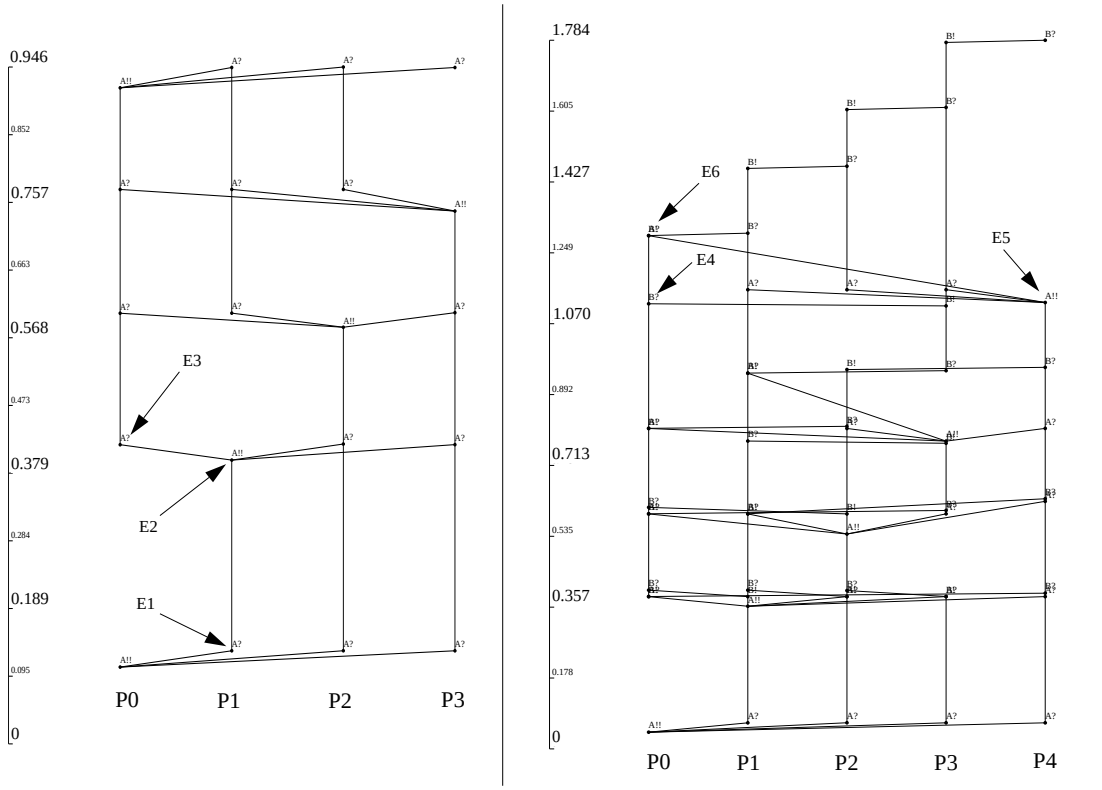


FIG. 4.5 - *Comparaison du comportement temporel observé avec la distribution « idéale » (à gauche), et avec une distribution « non idéale » (à droite)*

qu'un calcul y est en cours. Par exemple, le message émis par P4 à l'intention de P0 (événement E5) n'est réceptionné par P0 (événement E6) qu'après que P0 ait terminé le calcul ayant débuté avec l'événement E4.

Les outils de collecte, de traitement et de visualisation des traces qui nous ont permis de produire les diagrammes de causalité et les diagrammes temporels reproduits dans ce paragraphe ont été développés par C. Bareau dans le cadre de son travail de thèse [12].

L'observation des exécutions réparties constitue à n'en pas douter un atout majeur pour le développement et la mise au point d'algorithmes parallèles, mais l'interprétation des traces d'exécution générées avec la POM demeure encore assez empirique. L'exploitation effective des traces d'exécutions réparties passe nécessairement par l'emploi de techniques d'interprétation plus systématiques. Le développement de telles techniques fait l'objet de plusieurs études au sein du projet Pampa.

4.3 Optimisation des communications

4.3.1 Motivation

On a décrit au chapitre 3 les mécanismes de transfert de blocs qui ont été implantés dans les classes descendant de DIST_MATRIX, et montré dans le paragraphe 4.2.3 comment ces mécanismes peuvent servir à paralléliser les calculs portant sur des matrices distribuées.

On montre ici que ces mécanismes présentent toutefois l'inconvénient d'amener à la création d'un très grand nombre d'objets aux cours des échanges de données et qu'il est possible, au prix d'une programmation un peu plus fastidieuse ou d'une mise en œuvre différente des matrices distribuées, d'économiser sur le nombre d'objets créés lors des communications. On montre également qu'en exerçant un contrôle précis sur l'activité du ramasse-miettes, on peut obtenir une certaine forme de recouvrement calcul/communication aux cours des exécutions réparties.

4.3.2 Mécanismes alternatifs pour les transferts de blocs

En développant la classe DBLOCK_MATRIX, nous avons mis à profit la représentation interne sous forme de table de blocs matrices pour implanter deux fonctions *block* et *bring_block_to* permettant la transmission des blocs de partitionnement en diffusion ou en point-à-point (ces fonctions ont été décrites au paragraphe 3.4.1.4). On a également mis en œuvre suivant le même principe deux fonctions *column* et *bring_column_to* dans la classe DCOL_MATRIX, et deux fonctions *row* et *bring_row_to* dans la classe DROW_MATRIX (paragraphe 3.4.2).

Toutes ces fonctions de transfert de blocs permettent de faire abstraction des échanges de données lorsqu'on procède à la parallélisation d'un opérateur. Pourtant, elles ne sont pas totalement satisfaisantes, car leur utilisation amène à de trop nombreuses créations d'objets. En effet, chaque fois que la copie d'un bloc distant doit être amenée sur un nœud donné, on crée sur ce nœud destinataire un nouvel objet afin qu'il serve temporairement de réceptacle aux données reçues (voir par exemple le code de la fonction *bring_block_to* reproduit dans l'exemple 3.13 à la page 86).

Créer des objets en grand nombre au cours d'une phase de calcul ne constitue pas un réel problème tant que l'on ne s'intéresse qu'au bon comportement de l'application. En effet, aussi longtemps que le nombre d'objets référencés dans l'application demeure raisonnable, le ramasse-miettes est capable de détecter les objets obsolètes et de les supprimer, récupérant ainsi l'espace mémoire qu'ils occupaient. Cependant, créer une multitude d'objets temporaires au cours d'une exécution en-

traîne inévitablement une dégradation des performances globales de l'application, due notamment au fait que le ramasse-miettes doit être activé plus souvent et avoir des périodes d'activité plus longues.

Dans les deux paragraphes qui suivent, on présente deux méthodes permettant de réduire le nombre d'objets temporaires créés lors des réceptions de blocs dans les algorithmes parallèles impliquant des objets de type DBLOCK_MATRIX[17]. Ces méthodes ont toutes deux été expérimentées dans Paladin.

4.3.2.1 Gestion explicite d'objets « tampons »

Principe

Pour éviter de créer systématiquement un nouvel objet chaque fois que la copie d'un bloc distant doit être maintenue localement par un nœud, on peut choisir de renoncer aux abstractions offertes par les fonctions *block* et *bring_block_to*, et gérer alors explicitement les émissions et réceptions dans les algorithmes parallèles. En procédant de la sorte, on est en mesure de créer des matrices locales servant d'objets « tampons » pour maintenir localement des copies de blocs distants. On peut alors faire en sorte qu'une matrice tampon serve plusieurs fois au cours de l'exécution.

Illustration

Nous avons par exemple écrit une variante de l'opérateur *mult_dblock_dblock* selon cette idée. La nouvelle mise en œuvre de cet opérateur est reproduite dans l'exemple 4.7. Au lieu d'invoquer simplement les fonctions *block* et *bring_block_to* pour créer une copie locale d'un bloc distant, on gère à présent explicitement la localisation des blocs ainsi que leur transmission (lignes 22, 25, 31 et 37). En outre, lorsque les données d'un bloc distant doivent être réceptionnées et stockées localement, on ne crée plus systématiquement un nouvel objet de type LOCAL_MATRIX comme c'était le cas avec les fonctions *block* et *bring_block_to*, mais on réutilise l'espace de stockage constitué par deux objets tampons référencés par *A_i_buf* et *B_j_buf* (lignes 24 et 37). Ces deux objets sont créés une fois pour toutes avec les dimensions requises au début de l'exécution de l'opérateur *mult_dblock_dblock* (lignes 15 et 16), et le ramasse-miettes ne les collectera pas avant qu'ils soient dé-référencés, c'est-à-dire pas avant la fin du calcul.

On pourra noter qu'en choisissant de créer les deux objets *A_i_buf* et *B_j_buf* une fois pour toutes au début de l'exécution de *mult_dblock_dblock*, on restreint du même coup le domaine d'application de cet opérateur. En effet, pour que l'algorithme de

17. Il va de soi que ces méthodes peuvent aussi s'appliquer lorsque les « blocs » considérés sont en fait des vecteurs locaux résultant du partitionnement de matrices distribuées par lignes ou par colonnes.

Exemple 4.7

```
class DBLOCK_MATRIX inherit
   ...
feature -- Optimized operators
   mult_dblock_dblock (A: DBLOCK_MATRIX; B: DBLOCK_MATRIX) is
         -- C <- A * B, with C = Current.                                    5
      require
         -- same conditions as in former mult_dblock_dblock
         part_ok: ((A.nrow \\ A.dist.bfi) = 0)
                  and ((B.ncolumn \\ B.dist.bfj) = 0);
      local                                                                  10
         i, j: INTEGER;
         C_ij, A_i, B_j, A_i_buf, B_j_buf: LOCAL_MATRIX;
      do
        !!A_i_buf.make (A.dist.bfi, A.ncolumn);
        !!B_j_buf.make (B.nrow, B.dist.bfj);                                 15

         from i:= 0 until i > A.dist.nbimax loop
            if (POM.node_id = A.owner_of_block (i, 0)) then
               -- Broadcast A(i,:)
               A_i:= A.local_block (i, 0);                                   20
               A_i.bcast;
            else -- Receive A(i,:) from broadcast
               A_i:= A_i_buf;
               A_i.recv_bcast_from (A.owner_of_block (i, 0));
            end; -- if                                                       25
            from j:= 0 until j > B.dist.nbjmax loop
               if (POM.node_id = B.owner_of_block (0, j)) then
                  B_j:= B.local_block (0, j);
                  if (POM.node_id /= owner_C_ij) then -- Send B(:,j)
                     B_j.send (owner_of_block (i, j));                       30
                  end; -- if
               end; -- if
               if (POM.node_id = owner_of_block (i, j)) then
                  if (POM.node_id /= B.owner_of_block (0, j)) then
                     -- Receive B(:,j)                                       35
                     B_j:= B_j_buf; B_j.recv_from (B.owner_of_block (0, j));
                  end; -- if
                  -- Compute C(i,j) = A(i,:) * B(:,j)
                  C_ij:= local_block (i, j);
                  C_ij.mult (A_i, B_j);                                      40
               end; -- if
               j:= j + 1;
            end; -- loop j
            i:= i + 1;
         end; -- loop i                                                      45
      end; -- mult_dblock_dblock
   ...
end -- DBLOCK_MATRIX
```

l'exemple 4.7 puisse être utilisé il est nécessaire que les matrices opérandes soient partitionnées en blocs parfaitement homogènes[18]. La précondition introduite au niveau des lignes 9 et 10 précise cette contrainte supplémentaire imposée aux matrices opérandes de l'opérateur *mult_dblock_dblock*.

Discussion

La méthode consistant à gérer explicitement les échanges de données et les objets tampons permet d'améliorer de manière sensible les performances d'un algorithme parallèle. Le gain demeure cependant assez difficile à évaluer, car il dépend de la taille totale de l'application et de la quantité de mémoire disponible sur chaque nœud de la plate-forme parallèle utilisée (le ramasse-miettes fonctionne de manière incrémentale : il ne commence vraiment à rechercher les objets obsolètes que lorsque la mémoire approche de la saturation). Toutefois, il suffit d'observer le code de l'exemple 4.7 pour constater que cette approche n'est pas des plus élégantes. Le nouveau code de la procédure *mult_dblock_dblock* est beaucoup plus long et plus dense que celui reproduit dans l'exemple 4.6. La gestion explicite des objets tampons permet donc d'obtenir des algorithmes plus performants, mais moins lisibles. En outre, le codage de ces algorithmes est plus difficile : on doit créer judicieusement les objets tampons, et les utiliser à bon escient dans le cadre d'opérations d'émission et de réception explicites.

En dépit de ces inconvénients, la plupart des opérateurs parallèles que nous avons implantés dans les classes de Paladin l'ont été en utilisant cette technique. La méthode présentée dans le paragraphe suivant a néanmoins également été testée. Elle a permis d'observer des performances à peu près équivalentes.

4.3.2.2 Incorporation temporaire dans la table des blocs

Principe

Dans la version actuelle de Paladin, les matrices distribuées sont représentées en mémoire sous la forme d'une table de blocs. Chaque entrée non vide dans la table des blocs référence un bloc local. Une entrée vide dans la table indique simplement que le processeur local n'est pas propriétaire du bloc considéré (voir à ce sujet le paragraphe 3.4).

Dans la méthode proposée ici, on utilise les entrées jusqu'à présent inutilisées dans la table des blocs pour maintenir une référence vers l'éventuelle image locale

18. On rappelle que la classe DISTRIBUTION_2D n'impose pas que les facteurs de partitionnement *bfi* et *bfj* soient des diviseurs des dimensions *nrow* et *ncolumn*. En conséquence les blocs situés le long du « bord droit » ou du « bord inférieur » d'une matrice distribuée peuvent éventuellement être plus petits que les autres blocs (le lecteur pourra se référer au besoin au paragraphe 3.2.2).

d'un bloc distant. La sémantique de la fonction *local_block*, qui retourne la valeur d'une entrée particulière de la table des blocs, se trouve alors quelque peu altérée :

- Si le bloc (i, j) est un bloc local (c'est-à-dire un bloc dont le processeur local est propriétaire), alors l'expression *local_block(i, j)* retourne une référence vers ce bloc ;
- Sinon l'expression *local_block(i, j)* retourne soit *Void* (ce qui signifie qu'aucune image du bloc considéré n'est disponible localement), soit une référence vers une copie locale de ce bloc.

Mise en œuvre

Pour assurer la transmission des blocs et le maintien à jour de la table des blocs, nous avons doté la classe DBLOCK_MATRIX d'une procédure *refresh_block* mise en œuvre comme illustré dans l'exemple 4.8.

Exemple 4.8

```
refresh_block (i, j: INTEGER) is
    require
        valid_block: dist.valid_block (i, j);
    local
        new_block: like local_block;
    do
        if dist.block_is_local(i, j) then
            -- Broadcast local block(i,j) to all nodes
            local_block (i, j).bcast;
        else
            if (local_block (i, j) = Void) then
                -- There's no block in table of blocks. Create a new one.
                !!new_block.make (dist.limax(i), dist.ljmax(j));
                put_local_block (new_block, bi, bj);
            end; -- if
            -- Receive block(i,j) from owner of block
            local_block (i, j).recv_bcast_from (dist.owner_of_block (i, j));
        end; -- if
    ensure
        local_block: block_is_local(i,j) implies (Result = local_block(i, j));
    end; -- refresh_block
```

Lorsque la procédure *refresh_block* est invoquée dans une phase d'exécution SPMD, le nœud propriétaire du bloc (i, j) désigné diffuse l'information contenue

dans ce bloc à l'intention de tous les autres nœuds (ligne 9). Les autres nœuds consultent quant à eux leur table des blocs afin de voir si une copie du bloc (i, j) a déjà été stockée localement (ligne 11). Dans la négative, un nouveau bloc — instance de LOCAL_MATRIX — est créé localement et placé dans la table des blocs (lignes 13 et 14). Les données diffusées par le propriétaire du bloc (i, j) peuvent alors être réceptionnées, le bloc référencé par l'entrée (i, j) servant de zone tampon pour stocker ces données (ligne 17).

La routine *refresh_block* assure le rafraîchissement simultané du bloc (i, j) sur tous les nœuds participant à l'exécution. Nous avons également défini une routine *refresh_block_on* (dont la mise en œuvre n'est pas détaillée ici) assurant le rafraîchissement du bloc désigné sur un nœud particulier.

Si l'on se contentait, lors de l'exécution d'un algorithme parallèle, d'invoquer les routines *refresh_block* et/ou *refresh_block_on* pour amener et maintenir sur certains nœuds des copies locales de blocs distants, on risquerait de saturer rapidement la mémoire des nœuds participant aux calculs. En effet, des copies locales des blocs distants seraient créées, mais jamais détruites par le ramasse-miettes puisque toujours référencées au moins par la table des blocs (même si l'on perdait toute référence à un bloc donné au niveau d'un algorithme parallèle quelconque, ce bloc serait toujours référencé par la table des blocs et ne pourrait donc jamais être récupéré par le ramasse-miettes). On courrait alors le risque d'aboutir à la situation paradoxale où chaque nœud disposerait localement d'une image complète (mais pas forcément à jour) de chacune des matrices distribuées manipulées dans une application.

Pour éviter ce problème, l'idéal serait de disposer en Eiffel du mécanisme dit des « pointeurs faibles » (*weak pointers*), communément employé dans le domaine des bases de données à objets. Ce mécanisme permet d'obtenir qu'un objet dont on n'a pas un besoin impératif immédiat soit cependant maintenu en mémoire aussi longtemps que le ramasse-miettes n'est pas contraint de le supprimer pour récupérer de l'espace mémoire. Dans la pratique, le mécanisme des pointeurs faibles nécessite qu'à chaque objet soit associé un identificateur unique (en général une valeur entière) et qu'on puisse en passant cette valeur à une routine *ad hoc* récupérer, soit l'objet correspondant si celui-ci est encore disponible en mémoire, soit une référence nulle (*Void*) indiquant que cet objet a dû être supprimé par le ramasse-miettes.

Le mécanisme des pointeurs faibles ne pourrait être mis en œuvre au niveau du langage Eiffel, car si l'on gérait à ce niveau une table permettant d'associer un identificateur à chaque objet , on interdirait du même coup au ramasse-miettes de récupérer les objets référencés dans cette table. Le mécanisme des pointeurs faibles ne peut donc être mis en œuvre qu'à l'extérieur du langage Eiffel. Il serait possible d'implanter la table de correspondance en C et de développer une interface adé-

quate pour la manipuler depuis Eiffel, mais cette implantation serait certainement très fastidieuse. En fait, R. Bielak a récemment proposé dans [19] que le mécanisme des pointeurs faibles soit intégré dans la norme du langage Eiffel. Puisque ce mécanisme serait d'une grande utilité dans la mise en œuvre de la bibliothèque Paladin, mais aussi dans celle d'autres bibliothèques parallèles conçues selon la même approche, nous ne pouvons qu'approuver la suggestion de R. Bielak et espérer que le comité de normalisation NICE[19] en adoptera l'idée.

À défaut de disposer du mécanisme des pointeurs faibles dans la version actuelle d'Eiffel, et pour éviter malgré tout le risque de saturation de la mémoire évoqué plus haut, nous avons adjoint à la routine *refresh_block* une seconde routine ayant un rôle tout à fait opposé.

Exemple 4.9

```
forget_block (i, j: INTEGER) is
    require
        valid_block: dist.valid_block (i, j);
    do
        if (not dist.block_is_local(i, j)) then                          5
            -- Block is not local. Remove block from table of blocks.
            put_local_block (Void, bi + 1, bj + 1);
        end; -- if
    ensure
        block_is_local: block_is_local(i,j) implies (local_block(i, j) /= Void);   10
        block_not_local: (not block_is_local(i,j))
                            implies (local_block(i, j) = Void);
    end; -- forget_block
```

La routine *forget_block* reproduite dans l'exemple 4.9 permet de faire en sorte que les nœuds qui ne sont pas propriétaires d'un bloc particulier « oublient » l'image locale qu'ils peuvent éventuellement avoir de ce bloc. Concrètement, lorsque la routine *forget_block* est invoquée avec un couple de paramètres i et j, chaque nœud — à l'exception du nœud propriétaire du bloc désigné — examine l'entrée correspondante dans sa table des blocs et, si cette entrée est non vide, déréférence l'objet associé.

19. NICE : *Nonprofit International Consortium for Eiffel.* Comité chargé de superviser la norme du langage Eiffel et des bibliothèques standard associées.

Parallélisation des algorithmes

En utilisant judicieusement les routines *refresh_block* et *forget_block* décrites plus haut, on peut aisément développer des algorithmes parallèles dans lesquels les créations d'objets tampons sont maintenues au strict minimum, sans qu'il y ait pour autant le risque de saturer la mémoire disponible sur chaque nœud.

On a reproduit dans l'exemple 4.10 le code de la routine *mult_dblock_dblock* tel que nous l'avons redéfini dans un but d'expérimentation.

On notera que les blocs de la matrice opérande référencée par A sont rafraîchis sur tous les nœuds grace à l'invocation de la routine *refresh_block* (ligne 14). Les blocs de la matrice opérande référencée par B sont en revanche rafraîchis sur certains nœuds seulement grâce à l'emploi de la routine *refresh_block_on* (ligne 17). Pour calculer le produit des blocs matrices on n'utilise plus en lignes 20 et 21 que des accesseurs locaux. La routine *forget_block* est invoquée sur les matrices A et B en lignes 23 et 26 de telle manière que chaque nœud « oublie » les images locales des blocs distants dès qu'il n'en a plus l'usage.

Si le mécanisme des pointeurs faibles était disponible en Eiffel, la mise en œuvre de la routine *mult_dblock_dblock* serait semblable à celle de l'exemple 4.10, excepté que les invocations de la routine *forget_block* n'apparaîtraient pas.

4.3.3 Recouvrement calcul/communication

Les ramasse-miettes intégrés aux exécutifs des environnements Eiffel actuellement disponibles dans le commerce (*i.e. ISE, Tower, SIG*) mettent en œuvre les techniques de gestion de la mémoire les plus modernes. Ils peuvent être contrôlés de manière très précise grâce à un ensemble de routines offertes par la classe MEMORY, l'une des nombreuses classes de la bibliothèque Eiffel standard.

Dans EPEE, nous avons utilisé ces routines pour exercer un contrôle précis sur le ramasse-miettes afin d'obtenir un recouvrement efficace de l'activité du ramasse-miettes et des communications. Sur chaque nœud de la plate-forme parallèle utilisée, le ramasse-miettes est désactivé pendant les phases de calcul et réactivé de manière incrémentale lorsque le nœud est bloqué en attente de la réception d'un message. Ce mécanisme est illustré dans l'exemple 4.11, où on montre la mise en œuvre de la routine *recv_from*, l'une des primitives de réception offertes par la classe POM.

Dans le code de la routine *recv_from*, on invoque la routine *collect* de la classe MEMORY pour déclencher pendant l'attente d'un message un cycle de ramassage partiel des objets obsolètes. Il est à noter que les nombreuses routines de la classe MEMORY permettraient d'affiner encore le contrôle du ramasse-miettes (contrôle de la période de réactivation du ramasse-miettes, etc.).

Exemple 4.10

```
class DBLOCK_MATRIX inherit
   ...
feature -- Optimized operators
   mult_dblock_dblock (A, B : DBLOCK_MATRIX) is
         -- C <- A * B, with C = Current.                              5
      require
         -- size_ok: (same condition as default_mult)
         -- dist_ok: (please refer to the text)
      local
         i, j : INTEGER;                                              10
      do
         from i := 0 until i > A.dist.nbimax loop
            -- Refresh A(i,:)
            A.refresh_block (i, 0);
            from j := 0 until j > B.dist.nbjmax loop                  15
               -- Refresh B(:,j) on the owner of C(i,j)
               B.refresh_block_on (0, j, owner_of_block (i, j));
               if (block_is_local (i, j)) then
                  -- Compute locally C(i,j) = A(i,:) * B(:,j)
                  local_block (i, j).mult (A.local_block (i, 0),      20
                                           B.local_block (0, j));
               end; -- if
               B.forget_block (0, j);
               j := j + 1;
            end; -- loop j                                            25
            A.forget_block (i, 0);
            i := i + 1;
         end; -- loop i
      end; -- mult_dblock_dblock
   ...                                                                30
end -- DBLOCK_MATRIX
```

Exemple 4.11

```
expanded class POM
    ...
feature
    recv_from (pid: INTEGER; obj: ANY) is
        local                                                    5
            MEM: expanded MEMORY;
        do
            from -- Now on...
            until -- ... a message has been received
                (probe_from(pid) = 1)                            10
            loop
                -- Force a partial collection cycle if the garbage
                -- collector is enabled; do nothing otherwise.
                MEM.collect;
            end; -- loop                                         15
            -- Receive the message
            c_recv_from(pid, obj);
        end; -- recv_from
    ...
end -- POM                                                       20
```

4.4 Sélection dynamique des opérateurs

4.4.1 Motivation

L'optimisation — et en particulier la parallélisation — des opérateurs permettant de manipuler les agrégats matrices et vecteurs entraîne l'apparition de multiples routines encapsulant des mises en œuvre spécialisées de ces opérateurs. On a vu ainsi apparaître dans les classes descendant de la classe MATRIX des opérateurs spécialisés, capables de réaliser des calculs de manière efficace lorsque l'objet courant et les éventuels objets passés en paramètres satisfont certaines exigences portant sur leur type (*e.g.* matrice distribuée ou locale) ou sur d'autres critères (*e.g.* dimensions, schémas de partitionnement ou de placement, etc.).

Le langage Eiffel ne permettant pas de définir dans une classe plusieurs routines ayant le même nom, nous avons dû nous astreindre à donner des noms différents à toutes les routines encapsulées dans une même classe, même lorsque plusieurs de ces routines réalisent — du point de vue de l'utilisateur — le même calcul. Ainsi, dans la classe DBLOCK_MATRIX, nous avons dû maintenir simultanément deux routines *mult_default* et *mult_dblock_dblock*. La première est capable de calculer un produit de matrices de manière séquentielle quels que soient les types des trois matrices impliquées dans le calcul. La seconde, en revanche, ne peut calculer le produit de matrices que si les matrices sont toutes trois de type DBLOCK_MATRIX, et satisfont en outre certaines exigences portant sur leur schéma de partitionnement.

Il ne serait pas satisfaisant d'imposer à l'utilisateur de Paladin de désigner explicitement l'opérateur spécialisé le plus approprié pour réaliser un calcul. D'une part, ce serait renoncer à l'abstraction offerte par les classes MATRIX et VECTOR (on a en effet énoncé comme objectif au paragraphe 1.3 que toutes les matrices doivent pouvoir être manipulées de manière identique par l'utilisateur grâce à une interface homogène). D'autre part, ce serait obliger l'utilisateur à se tenir informé de l'apparition de nouveaux opérateurs spécialisés dans la bibliothèque. Ce document visant notamment à promouvoir le développement de bibliothèques optimisables et/ou parallélisables de manière incrémentale, il ne saurait être question d'imposer une telle charge à l'utilisateur.

L'optimisation et la parallélisation de la bibliothèque Paladin doivent donc impérativement demeurer transparentes pour l'utilisateur. On doit faire en sorte que celui-ci puisse manipuler des agrégats matrices et vecteurs en s'appuyant uniquement sur les informations contenues dans l'interface des classes MATRIX et VECTOR. Pour chaque opérateur invoqué au cours de l'exécution d'un programme d'application, la sélection de l'opérateur spécialisé le mieux adapté pour effectuer le calcul demandé doit être réalisée de manière automatique et transparente. Ceci ne pose pas les mêmes problèmes selon que ce calcul implique un seul objet ou plusieurs. On

discute de la sélection dynamique des opérateurs unaires dans le paragraphe 4.4.2, et de celle des opérateurs N-aires dans le paragraphe 4.4.3.

4.4.2 Cas des opérateurs unaires

Dans le cas des opérateurs n'admettant pas d'autre paramètre que l'objet courant (on parlera alors d'opérateurs unaires), le mécanisme de la liaison dynamique suffit pour garantir la sélection dynamique des opérateurs de manière totalement transparente pour l'utilisateur.

Considérons de nouveau l'exemple de l'opérateur fonction *trace*, dont une version parallèle a été décrite dans la paragraphe 4.2.3. Dès lors que l'on a redéfini dans la classe DIST_MATRIX la fonction *trace* en la dotant d'un algorithme optimisé (*i.e.* tenant compte de la distribution), le mécanisme de la liaison dynamique associé au langage Eiffel garantit que cet algorithme sera automatiquement sélectionné à l'exécution chaque fois que l'on invoquera la routine *trace* sur un objet de type conforme à DIST_MATRIX (c'est-à-dire tout objet instancié d'après la classe DIST_MATRIX ou d'après une classe descendant de DIST_MATRIX[20]). Considérons le petit programme SPMD suivant :

Exemple 4.12

```
local
    A, B : MATRIX;
    v, w : DOUBLE;
do
   !LOCAL_MATRIX !A.make (...);                                    5
   !DBLOCK_MATRIX !B.make (...);
    ...
    v := A.trace; -- The default algorithm is used
    w := B.trace; -- The parallel algorithm is used
    ...                                                            10
end;
```

En supposant que la routine *trace* n'a été redéfinie ni dans la classe LOCAL_MATRIX, ni dans la classe DBLOCK_MATRIX, c'est alors l'algorithme de calcul par défaut (celui de la classe MATRIX) qui va être exécuté pour calculer la trace de la matrice référencée par A. En effet, cet objet est de type LOCAL_MATRIX (l'entité A étant quant à elle de *type dynamique* LOCAL_MATRIX). La fonction *trace* n'ayant

20. On supposera ici que la fonction *trace* n'a été redéfinie dans aucune des classes descendant de DIST_MATRIX.

pas été redéfinie dans la classe LOCAL_MATRIX, c'est la mise en œuvre héritée de MATRIX qui « vaut » pour toutes les matrices locales.

L'objet référencé par B est lui de type DBLOCK_MATRIX. La fonction *trace* n'ayant pas été redéfinie dans la classe DBLOCK_MATRIX, c'est la mise en œuvre héritée de la classe DIST_MATRIX qui « vaut » pour cet objet. C'est donc bien l'algorithme parallèle encapsulé dans la fonction *trace* de la classe DIST_MATRIX qui va être exécuté pour calculer la trace de la matrice référencée par B.

L'exemple 4.12 montre que le mécanisme de liaison dynamique assure la sélection transparente des opérateurs unaires du point de vue du programmeur d'application. Lorsqu'un opérateur unaire est invoqué sur une matrice distribuée et qu'une mise en œuvre parallèle de cet opérateur a été intégrée dans la bibliothèque Paladin, l'algorithme parallèle est exécuté automatiquement à la place de l'algorithme séquentiel par défaut décrit dans la classe MATRIX.

Coût de la liaison dynamique

Dans les langages à objets à typage statique offrant uniquement l'héritage simple (*e.g.* Modula 3, Ada 95), le mécanisme de la liaison dynamique peut en général être implanté de manière à s'exécuter en temps constant. Il n'en va pas toujours de même avec les langages offrant le mécanisme d'héritage multiple. Dans le cas du langage C++, par exemple, le temps requis pour effectuer la liaison dynamique suite à l'invocation d'une routine peut être variable en cas d'héritage multiple. Avec les compilateurs Eiffel actuels, la sélection de la routine devant effectivement être exécutée nécessite simplement deux indirections successives, qu'il y ait héritage simple ou héritage multiple. La liaison dynamique s'effectue donc toujours en temps constant. En outre, les compilateurs Eiffel sont capables de détecter les invocations de routines pour lesquelles l'emploi de la liaison dynamique n'est pas requise et de remplacer alors la liaison dynamique par un simple appel de procédure (cette caractéristique a déjà été évoquée dans le paragraphe 1.3.2.3).

4.4.3 Cas des opérateurs N-aires

Le problème

Le mécanisme de la liaison dynamique mis en œuvre dans le langage Eiffel[21] est parfois qualifié de mécanisme de « sélection dynamique simple » (ou *single dispatching*), car il base la liaison dynamique sur le type du seul objet courant, c'est-à-dire l'objet référencé par l'entité désignée à gauche du point dans une expression pointée

21. Ainsi d'ailleurs que dans les langages C++, Modula 3, Ada 95, etc.

telle que *A.trace* ou *A.add(B)*. Ce mécanisme n'est donc pas suffisant pour assurer la sélection dynamique des opérateurs de Paladin impliquant plusieurs opérandes, car il est parfois indispensable que les types des objets passés en paramètres soient pris en compte lors de la sélection.

L'idéal serait que la sélection de l'algorithme approprié soit réalisée automatiquement à l'exécution en fonction des types de tous les paramètres impliqués dans un calcul. Dans les langages à objets fortement typés tels que Eiffel, C++, Ada95 ou Modula3, un tel mécanisme de sélection dynamique multiple (*multiple dispatching*) n'est pas disponible. Seul existe le mécanisme de sélection dynamique simple, dont on a montré au paragraphe précédent qu'il convient parfaitement pour assurer la sélection dynamique des opérateurs unaires. Lorsqu'il s'applique aux opérateurs N-aires, cependant, le mécanisme de sélection dynamique simple introduit une certaine asymétrie entre les objets impliqués dans un calcul dans la mesure où il privilégie l'objet « courant » et ne tient aucun compte des autres objets passés en paramètres à l'opérateur.

Dans les deux paragraphes qui suivent, on montre qu'il est possible d'implanter « manuellement » la sélection dynamique multiple lorsqu'on n'utilise qu'un langage à sélection dynamique simple. Dans le paragraphe 4.4.3.1, on présente la technique dite de « sélection en cascade », qui a pour avantage de ne s'appuyer que sur le seul mécanisme de la liaison dynamique simple. Dans le paragraphe 4.4.3.2, on présente une technique alternative, qui n'est applicable que lorsque le langage à objets utilisé permet de tester le type d'un objet à l'exécution. Cette dernière technique est celle que nous avons retenue pour réaliser la sélection dynamique multiple dans les classes de Paladin. On discute enfin dans le paragraphe 4.4.3.3 de ce que pourraient nous apporter des langages à sélection dynamique multiple tels que CLOS, Cecil, etc.

4.4.3.1 Technique de la « sélection en cascade »

Le mécanisme de la *sélection en cascade* constitue une manière assez élégante de réaliser la sélection dynamique multiple des opérateurs avec un langage ne disposant que du mécanisme de sélection dynamique simple.

Principe

Le mécanisme de la sélection en cascade peut être décrit succinctement de la manière suivante : partant du constat que lors de l'exécution d'un opérateur, seul le type de l'objet courant est connu avec une précision satisfaisante (ce type étant caractérisé par la classe dans laquelle l'opérateur est défini[22]), le principe de la sélection en cascade consiste à définir un ensemble de routines qui s'appellent les

22. Ou dans certains cas par l'une des classes descendant de celle où est défini l'opérateur.

unes les autres « en cascade » en faisant subir un décalage à leurs paramètres formels à chaque appel de manière à ce que chacun des paramètres formels prenne lors d'un appel la position de l'objet courant. On peut ainsi identifier peu à peu le type de tous les paramètres formels en s'appuyant sur le seul mécanisme de la sélection dynamique simple, et invoquer finalement l'opérateur *ad hoc* lorsque le type de chacun des paramètres formels a pu être identifié.

Illustration

Prenons l'exemple de l'addition de deux matrices. Le problème est de choisir dynamiquement entre les opérateurs *add_default* et *add_dblock* lorsque l'opération *A.add(B)* est invoquée alors que l'objet référencé par l'entité A est de type DBLOCK_MATRIX (l'objet référencé par B étant en revanche de type indéterminé).

Lorsque l'expression *A.add(B)* est insérée dans un programme SPMD et que A est de type dynamique DBLOCK_MATRIX, la sélection dynamique simple assure que c'est l'opérateur *add* défini dans la classe DBLOCK_MATRIX qui va être exécuté. Au cours de l'exécution de cet opérateur, on se trouve dans la situation suivante : le type de l'opérande A est connu avec précision (A étant l'objet courant, son type est décrit par la classe englobant l'opérateur en cours d'exécution, c'est-à-dire la classe DBLOCK_MATRIX), alors qu'on ne sait rien du type de l'objet référencé par B (sinon qu'il est d'un type conforme à MATRIX). Dans le corps de la routine *add* définie dans la classe DBLOCK_MATRIX, on délègue à l'objet référencé par B la responsabilité de la suite de l'exécution en plaçant dans le corps de la routine *mult* une expression de la forme *B.rev_add_dblock(A)*.

La sémantique de la routine *rev_add_dblock* est un peu différente de celle de la routine *add*. En effet, l'expression *A.add(B)* signifie *grosso modo* « ajouter la matrice référencée par B à la matrice référencée par A[23], quels que soient les types dynamiques de A et de B », alors que l'expression *A.rev_add_dblock(B)* signifie « ajouter la matrice A à la matrice B, cette dernière ayant été identifiée comme étant de type DBLOCK_MATRIX ».

Lors de l'invocation de l'opérateur *rev_add_dblock*[24], la sélection dynamique simple va de nouveau intervenir, mais en s'appliquant cette fois à l'objet B. Dans le corps de la routine *rev_add_dblock* finalement sélectionnée, il n'y a plus aucune ambiguïté sur le type des deux matrices opérandes, le type de A étant codé explicitement dans le nom de la routine et celui de B étant caractérisé par la classe dans lequel la routine *rev_add_dblock* est définie.

23. A et B étant de type statique MATRIX.

24. Le préfixe *rev_* vise à attirer l'attention du lecteur de la routine *rev_add_dblock* sur le fait que cette routine fonctionne « à l'envers », ajoutant l'objet courant à l'objet passé en paramètre et non l'inverse.

On a reproduit ci-dessous les opérateurs *add* et *rev_add_dblock* tels qu'ils pourraient être définis dans les classes MATRIX et DBLOCK_MATRIX de Paladin.

Exemple 4.13

```
deferred class MATRIX inherit
    ...
feature -- Operators
    add, add_default (B: MATRIX) is
        -- Add matrix B to 'Current'                                    5
        do ... end;
    rev_add_dblock (B: DBLOCK_MATRIX) is
        -- Add 'Current' to matrix B (whose dynamic type is known)
        do
            B.add_default(Current);                                     10
        end;
    ...
end -- class MATRIX
```

On notera qu'afin de ne pas laisser la routine *rev_add_dblock* différée dans la classe MATRIX on se contente de la définir sur la base d'un appel à la routine *add_default* (en prenant soin toutefois de permuter les deux opérandes lors de cet appel).

Ayant ajouté la routine *rev_add_dblock* dans MATRIX, on peut alors redéfinir les routines *add* et *rev_add_dblock* dans la classe DBLOCK_MATRIX (la routine *add_dblock* étant quant à elles définie comme on l'a déjà montré au paragraphe 4.2.3).

Vision de l'utilisateur

Avec la technique d'implantation présentée ci-dessus, on assure bien la sélection dynamique multiple des opérateurs en garantissant une transparence totale pour le programmeur d'application. Il suffit pour s'en convaincre de considérer le petit programme d'application de l'exemple 4.15.

Dans cet exemple on crée trois matrices — que l'on supposera de même taille — que l'on affecte respectivement aux variables locales A, B et C. La matrice référencée par A est une matrice locale (la variable A prend donc le type dynamique LOCAL_MATRIX) alors que les matrices référencées par A et B sont distribuées par blocs (les variables A et B prennent donc le type dynamique DBLOCK_MATRIX).

Lorsqu'on invoque l'opérateur d'addition sur la variable B, le mécanisme de la liaison dynamique nous assure que c'est la routine définie dans la classe DBLOCK_MATRIX qui va être exécutée. Au cours de cette exécution, la routine *rev_add_dblock*

Exemple 4.14

```
class DBLOCK_MATRIX inherit
    DIST_MATRIX
        redefine
            add, rev_add_dblock
        end
    ...
feature -- Optimized operators
    add (B: MATRIX) is
        do
            B.rev_add_dblock (Current);
        end;
    rev_add_dblock (B: DBLOCK_MATRIX) is
        do
            if (dist.dist.bfi = B.dist.bfi)  -- Test if same block size
                and then (dist.bfj = B.dist.bfj) then
                B.add_dblock (Current);
            else
                B.add_default (Current);
            end; -- if
        end;
    add_dblock (B: DBLOCK_MATRIX) is
        do
            -- Algorithm defined as shown previously
        end;
    ...
end -- class DBLOCK_MATRIX
```

Exemple 4.15

```
local
    A, B, C : MATRIX;
do
   !LOCAL_MATRIX !A.make (...);
   !DBLOCK_MATRIX !B.make (...);                                    5
   !DBLOCK_MATRIX !C.make (...);
    ...
    B.add (A);      -- 'add_default' is selected

    B.add (C);      -- 'add_dblock' is selected if B and C          10
                    -- have the same block size. Otherwise
                    -- 'add_default' is selected
end ;
```

(voir exemple 4.14) va être invoquée sur la variable A, avec la variable B en paramètre. A ayant pour type dynamique le type LOCAL_MATRIX, le mécanisme de la liaison dynamique va intervenir une fois encore pour faire en sorte que la routine *rev_add_dblock* de la classe LOCAL_MATRIX (c'est-à-dire la routine par défaut héritée de la classe MATRIX, voir exemple 4.13) soit exécutée. Au cours de l'exécution de cette routine, la routine *add_default* de la classe MATRIX va finalement être invoquée sur la variable B (avec A en paramètre). Au terme de la sélection dynamique en cascade, c'est donc bien l'algorithme séquentiel par défaut qui va être exécuté pour ajouter la matrice A à la matrice B.

Lorsqu'on invoque ensuite l'opérateur d'addition sur la variable B avec en paramètre la variable C, la routine *add* de la classe DBLOCK_MATRIX va de nouveau être sélectionnée grâce à la liaison dynamique. Cette fois, cependant, la matrice C passée en paramètre est de type DBLOCK_MATRIX. Lors de l'invocation de *rev_add_dblock* sur C, c'est donc la routine de la classe DBLOCK_MATRIX qui va être exécutée (voir exemple 4.14). Au cours de son exécution, les facteurs de partitionnement des deux matrices distribuées B et C vont être comparés, et en fonction du résultat de cette comparaison l'une des routines *add_dblock* ou *add_default* va être invoquée pour réaliser le calcul requis.

Discussion

L'avantage principal du mécanisme de sélection en cascade est qu'il peut être appliqué avec n'importe quel langage à objets à typage statique. La technique de sélection en cascade constitue donc un atout majeur lorsqu'on désire réaliser la sélec-

tion dynamique multiple de certains opérateurs et que le langage utilisé ne permet pas de tester explicitement le type dynamique des entités référençant les objets[25]. Cependant, la sélection en cascade demeure assez lourde à mettre en œuvre. Outre qu'elle introduit de nouvelles dépendances entre les classes de la hiérarchie et oblige à modifier plusieurs de ces classes simultanément (dans l'exemple précédent on a dû modifier à la fois la classe MATRIX et la classe DBLOCK_MATRIX), cette technique oblige à de fastidieuses permutations des paramètres afin que l'objet dont on désire identifier le type devienne temporairement l'objet « courant ». La technique de sélection en cascade induit en outre une explosion du nombre de routines devant être implantées dans les classes de la hiérarchie.

Pour toutes ces raisons, il n'est guère envisageable d'appliquer cette technique pour sélectionner des opérateurs ayant plus de deux opérandes (*i.e.* l'objet « courant » plus un objet passé en paramètre). Pour les opérateurs binaires, en revanche, le schéma général de la sélection dynamique multiple se ramène à un schéma plus simple de « sélection dynamique double » (*double dispatch*) qui a déjà fait l'objet d'études approfondies, notamment dans [75] où D. Ingalls introduit la sélection dynamique double à travers la notion de *polymorphisme multiple des expressions.*

Lorsque la technique de sélection en cascade est appliquée aux opérateurs binaires, la procédure de mise en œuvre est suffisamment systématique pour qu'on puisse envisager son automatisation. Ainsi, dans [72] K. Hebel et R. Johnson décrivent un « fouineur » (*browser*) permettant de gérer la mise en œuvre d'opérateurs arithmétiques binaires en Smalltalk-80. Cet outil se présente à l'utilisateur sous la forme d'un tableur à deux dimensions dans lequel chaque case correspond à une combinaison possible des types des opérandes de l'opérateur binaire considéré. L'utilisateur peut faire en sorte qu'à plusieurs cases corresponde une mise en œuvre commune de l'opérateur. L'outil est capable de générer automatiquement la plupart des routines assurant la sélection dynamique double, l'utilisateur n'ayant plus alors qu'à fournir les informations manquantes, c'est-à-dire les détails de mise en œuvre des routines de calcul proprement dites.

On pourrait envisager de développer un outil analogue au « fouineur » de Hebel et Johnson, capable de manipuler un ensemble de classes Eiffel afin d'y intégrer automatiquement des « multi-opérateurs » binaires (c'est-à-dire des opérateurs bénéficiant de la sélection dynamique double grâce à une mise en œuvre automatique de la sélection en cascade). Un tel outil contribuerait sans aucun doute à simplifier la tâche du programmeur désireux d'implanter des multi-opérateurs binaires dans certaines classes Eiffel, mais il ne suffirait cependant pas pour résoudre l'ensemble

25. C'est par exemple le cas du langage C++ : bien que la notion de type dynamique soit définie dans la norme du langage (sous l'appellation *RTTI: Runtime Type Identifier*), la plupart des compilateurs C++ actuels ne permettent pas de tester réellement le type dynamique d'un objet.

des problèmes rencontrés dans la mise en œuvre de Paladin. En effet, les opérateurs de Paladin ne sont pas limités aux seules opérations unaires et binaires. Certains opérateurs — bien qu'ils ne soient pas les plus nombreux — admettent trois, voire même quatre ou cinq paramètres dont le type dynamique doit être pris en compte lors de la sélection dynamique. Pour assurer la sélection dynamique de tels opérateurs N-aires, il est préférable que l'on soit en mesure de tester le type des objets au cours d'une exécution. Dans ce cas, on peut appuyer la sélection dynamique sur des tests explicites des types des objets, comme expliqué dans le paragraphe suivant.

4.4.3.2 Sélection dynamique par test explicite des objets

Principe

Dans la version actuelle de Paladin, on met en œuvre la sélection dynamique des opérateurs N-aires en mettant à profit le mécanisme de l'*essai d'affectation* (*assignment attempt*) qui combine en Eiffel les mécanismes d'affectation polymorphe et de test de conformité de type [97]. Plus précisément, on procède grâce à ces mécanismes à des tests explicites portant sur le types des objets passés en paramètres à un opérateur.

Illustration

Dans la classe DBLOCK_MATRIX, on redéfinit l'opérateur d'addition de manière à tester le type de l'objet passé en paramètre, comme illustré dans l'exemple 4.16.

Si le test révèle que le type de l'objet référencé par le paramètre formel B — c'est-à-dire le type dynamique de B — n'est pas conforme au type DBLOCK_MATRIX, alors l'algorithme par défaut hérité de la classe MATRIX (opérateur renommé en *add_default* dans la clause d'héritage) est exécuté. Si par contre il s'avère que l'objet référencé par B est de type DBLOCK_MATRIX (ou d'un type conforme à DBLOCK_MATRIX) et qu'en outre le partitionnement de cet objet est le même que celui de la matrice courante (c'est-à-dire que les blocs de partitionnement ont la même taille dans les deux matrices), alors c'est l'algorithme parallèle optimisé de l'opérateur *add_dblock* qui est invoqué.

Vision de l'utilisateur

Avec la technique d'implantation présentée ci-dessus, on assure bien la sélection dynamique multiple des opérateurs en garantissant une transparence totale pour le programmeur d'application. Il suffit pour s'en convaincre de considérer encore une fois le petit programme d'application de l'exemple 4.15.

Exemple 4.16

```
class DBLOCK_MATRIX inherit
   DIST_MATRIX
      rename
         add as add_default      -- Keep the default sequential operator
      end                                                                    5
   ...
feature -- Optimized operators

   add (B: MATRIX) is
      local                                                                  10
         tmp_B: DBLOCK_MATRIX;
      do
         tmp_B ?= B;     -- tmp_B := B if B conforms to 'DBLOCK_MATRIX'
                         -- tmp_B := Void elsewhere
         if (tmp_B /= Void)                                                  15
            and then (dist.bfi = tmp_B.dist.bfi)  -- Test if same block size
            and then (dist.bfj = tmp_B.dist.bfj) then
            add_dblock (tmp_B);
         else
            add_default (B);                                                 20
         end; -- if
      end;
   ...
end -- class DBLOCK_MATRIX
```

Lorsqu'on invoque l'opérateur d'addition sur la variable B, le mécanisme de la liaison dynamique nous assure cette fois encore que c'est la routine *add* définie dans la classe DBLOCK_MATRIX qui va être exécutée. Au cours de cette exécution, le test du type de l'entité A passée en paramètre à l'opérateur va révéler que A n'est pas de type dynamique DBLOCK_MATRIX (ni même d'un type conforme à DBLOCK_MATRIX). En conséquence la routine d'addition par défaut va finalement être exécutée.

Lorsqu'on invoque ensuite l'opérateur d'addition sur la variable B pour la seconde fois, la routine *add* de la classe DBLOCK_MATRIX va de nouveau être sélectionnée grâce à la liaison dynamique. Le test du type de l'entité C va cette fois révéler que C est bien de type dynamique DBLOCK_MATRIX. Par conséquent, les paramètres de partitionnement des deux matrices vont être comparés, et selon que les matrices ont ou non le même partitionnement, l'une des routines *add_default* ou *add_dblock* va finalement être sélectionnée.

Discussion

Il peut paraître fastidieux de devoir procéder à des tests explicites sur les types des paramètres passés à un opérateur avant d'invoquer ensuite explicitement l'opérateur ainsi sélectionné. On peut cependant énoncer les arguments suivants en faveur de cette approche :

- Le coût des tests réalisés pour sélectionner l'algorithme le plus approprié pour effectuer un calcul est en général totalement négligeable par rapport au coût du calcul proprement dit. Ainsi, dans l'exemple de l'addition de matrices, le coût des tests visant à choisir entre *add_default* et *add_dblock* demeure insignifiant comparé à la complexité des algorithmes de calcul encapsulés dans ces deux opérateurs [26].

- La méthode proposée pour effectuer la sélection dynamique des opérateurs permet de préserver une certaine localité des opérateurs optimisés. Ainsi, l'opérateur *add_dblock* ayant été encapsulé dans la classe DBLOCK_MATRIX, c'est dans cette même classe que la routine *add* doit être renommée en *add_default* et redéfinie afin d'intégrer les tests de conformité et les branchements *ad hoc*. Aucune autre classe de Paladin n'est donc perturbée par l'ajout de l'opérateur spécialisé *add_dblock*. Il n'en allait pas de même avec la technique de sélection en cascade présentée au paragraphe 4.4.3.1 : l'adjonction d'une routine telle

26. On suppose bien sûr que les vecteurs et matrices manipulés dans Paladin sont bien des agrégats, c'est-à-dire des objets de très grande taille. Si on devait ne manipuler avec Paladin que des matrices de très petite taille, le coût de la sélection dynamique multiple deviendrait nettement prohibitif. Mais dans ce cas, il n'y aurait de toute façon pas lieu de distribuer ces matrices.

que *rev_add_dblock* au niveau de la classe DBLOCK_MATRIX nous obligeait à insérer une routine par défaut de même nom dans la classe MATRIX. La méthode présentée ici est donc plus satisfaisante dans la mesure où elle ne fait pas apparaître de nouvelles dépendances entre les classes de la hiérarchie.

- Contrairement à la méthode de sélection en cascade, la méthode s'appuyant sur des tests explicites des types des objets peut s'appliquer quel que soit le nombre d'objets passés en paramètres à un opérateur. Il n'est en effet pas beaucoup plus difficile de tester un à un tous les objets passés en paramètre à une routine que d'en tester un seul. Alors que la technique de sélection en cascade amène très vite à une explosion combinatoire du nombre de routines devant être intégrées dans les classes de la hiérarchie dans le seul but d'assurer la sélection dynamique des opérateurs, la méthode présentée ici permet de concentrer tout l'effort de sélection dynamique d'un opérateur dans quelques routines (en pratique, on place une routine assurant la sélection dynamique dans chacune des classes où on a mis en œuvre une version optimisée de l'opérateur considéré).

Avant de clore ce paragraphe et d'évoquer les perspectives des travaux actuels visant au développement de langages à sélection dynamique multiple, il convient de noter que la méthode présentée ici, tout comme celle présentée au paragraphe 4.4.3.1, va au delà de ce qu'on entend traditionnellement par mécanisme de *multiple dispatch*. En effet la sélection peut ne pas s'effectuer uniquement en fonction des types dynamiques des paramètres. Dans le cas de l'addition de matrices, l'opérateur *add_dblock* est choisi pour effectuer les calculs si la matrice passée en paramètre est de type DBLOCK_MATRIX et si les schémas de partitionnement des deux matrices opérandes sont compatibles.

4.4.3.3 Vers des langages à sélection dynamique multiple

Plusieurs travaux ont déjà été menés qui visent à intégrer la sélection dynamique multiple dans un langage à objets. CLOS [47, 20] et son prédécesseur Common-Loops [21] peuvent être qualifiés de langages pionniers dans ce domaine. Au lieu de définir les méthodes (autre appellation des routines dans certains langages à objets) comme faisant partie de types de données abstraits, les *multi-méthodes* (méthodes pouvant avoir plusieurs signatures et plusieurs mises en œuvre alternatives) sont définies à l'extérieur des objets. Dans CLOS, les multi-méthodes ayant le même nom sont regroupées de manière à constituer des *fonctions génériques*. Ces fonctions sont mises en œuvre selon un schéma de traitement par cas qui s'apparente beaucoup à celui du *pattern matching* qui prévaut dans les langages fonctionnels (ceci n'a d'ailleurs rien d'étonnant puisque CLOS combine la programmation fonc-

tionnelle en Lisp avec certains mécanismes de la programmation par objets). Selon C. Chambers, cependant, l'utilisation de langages « hybrides » tels que CLOS risque d'encourager le programmeur à privilégier un style de programmation basé sur une décomposition fonctionnelle plutôt que sur l'abstraction des données [32]. Chambers propose donc une approche alternative pour développer des langages à objets à sélection dynamique multiple. Selon lui une multi-méthode doit être perçue, non plus comme une fonction générique n'appartenant à aucun type de donnée particulier, mais plutôt comme appartenant simultanément à tous les types de données impliqués dans la sélection dynamique de la méthode. En adoptant ce point de vue original sur les multi-méthodes, on est en mesure de préserver un style de programmation basé sur l'abstraction de données. Le langage Cecil [32] est un langage prototype conçu par Chambers suivant cette idée.

À ce jour, aucun des travaux visant à la mise au point d'un langage à objets à typage statique et à sélection dynamique multiple n'a encore permis d'aboutir à un produit commercial. D'ailleurs, même si un tel langage existait, il ne permettrait pas de résoudre tous les problèmes de sélection dynamique rencontrés dans Paladin. On a en effet montré avec l'exemple de l'addition de matrices que la sélection dynamique requise dans Paladin n'est pas toujours basée que sur le type des objets. D'autres critères doivent parfois être pris en compte lors de la sélection des opérateurs (taille des blocs de partitionnement, type de placement, etc.). Un langage incluant la notion de multi-méthode et doté d'un mécanisme de sélection dynamique multiple ne suffirait donc pas à résoudre tous nos problèmes, mais il contribuerait certainement à les diminuer. Quoiqu'il en soit, à défaut de disposer dès à présent d'un langage de ce type, on doit se contenter d'implanter « manuellement » la sélection dynamique multiple dans un langage à sélection dynamique simple tel que Eiffel, C++[27], Modula3 ou Ada95. Les deux techniques évoquées précédemment (la première basée sur la sélection en cascade, la seconde sur le test explicite du type des objets) nous ont permis d'assurer dans Paladin la sélection dynamique des opérateurs conformément à nos besoins, et de garantir ainsi une transparence maximale pour l'utilisateur de la bibliothèque.

27. Le mécanisme de *surcharge d'opérateurs* de C++, bien que souvent confondu par les utilisateurs de ce langage avec un mécanisme de sélection dynamique multiple, ne porte en fait que sur les types statiques des objets. La sélection est donc résolue à la compilation et non à l'exécution comme l'exigerait un véritable mécanisme de sélection dynamique.

4.5 Interfaçage avec le noyau BLAS

4.5.1 Motivations

La syntaxe du langage Eiffel rend possible l'appel de routines externes depuis n'importe quelle classe. Cette possibilité peut parfois se révéler particulièrement intéressante car elle permet d'interfacer une hiérarchie de classes avec des modules ou bibliothèques précompilés. On peut ainsi gagner en termes de temps de développement (ce qui a déjà été écrit n'a pas à être réécrit), mais souvent aussi en termes de performances, pour peu que le code externe ainsi utilisé ait été particulièrement optimisé.

La possibilité d'interfacer des classes avec du code externe est en fait l'un des atouts assez méconnus de la programmation par objets : le principe de la réutilisation de code ne se limite pas à la seule réutilisation de classes. Des modules et bibliothèques externes précompilés peuvent également être réutilisés dans le monde des objets, même s'ils n'ont pas été à l'origine développés dans cette optique. On peut associer à un module externe écrit en C, en Fortran, en Pascal, etc. une classe servant d'interface avec le langage à objets utilisé. On rend ainsi ce module externe aisément utilisable depuis ce langage, tout en appliquant le principe du masquage de l'information grâce au mécanisme de l'encapsulation.

Pour illustrer cette idée, la bibliothèque Paladin a été interfacée avec le noyau de calcul BLAS [88] (*Basic Linear Algebra Subroutines*), qui constitue une batterie de routines de calcul exprimées en Fortran pour l'algèbre linéaire. BLAS est en général implanté sur les machines par les constructeurs eux-mêmes et ce, la plupart du temps, directement en assembleur et en tenant compte des spécificités de la machine cible (on a déjà abordé ce sujet au paragraphe 1.3.1). En conséquence, les performances des routines BLAS sont souvent sans commune mesure avec celles que l'on peut obtenir à partir d'un code C ou Fortran compilé. Ainsi, alors que sur un processeur de type Sparc on observe des résultats à peu près similaires lorsqu'on compare les performances d'une routine BLAS avec celles d'une routine équivalente écrite « à la main » en C ou en Fortran, il n'en va pas de même avec un processeur de type Intel *i*860. Sur la machine parallèle Intel Paragon XP/S dont les nœuds comportent des processeurs *i*860, des mesures portant sur le calcul de produits de matrices 400×400 en double précision ont révélé que la routine `DGEMM` du noyau BLAS (calculant le produit de matrices de nombres réels en double précision) s'exécute environ 25 fois plus vite qu'un algorithme équivalent exprimé en C ou en Fortran[28]. Cette énorme différence s'explique par le fait que les compilateurs

28. Les mesures évoquées ici ont été réalisées en utilisant les compilateurs *icc* et *if77* version 4.5 produits par *Intel Corporation* et *The Portland Group* avec lors de chaque compilation l'option

actuels sont incapables de gérer de manière efficace la hiérarchie complexe de la mémoire sur les nœuds de la machine Paragon (sur chaque nœud de cette machine la mémoire est hiérarchisée en trois niveaux : les registres internes de l'*i*860, la mémoire cache, et la mémoire de *swap*). En revanche, les routines BLAS encapsulées dans la bibliothèque *libkmath.a* ont été construites par les ingénieurs de la société Intel de manière à exploiter au mieux les caractéristiques de la mémoire sur la machine Paragon (nombre de registres, taille de la mémoire cache, etc.).

Il eut été dommage de ne pas exploiter de telles performances dans Paladin, mais il aurait en même temps été inacceptable de limiter la souplesse, l'extensibilité et le confort d'utilisation de notre bibliothèque sous prétexte que les routines BLAS ne peuvent manipuler que des vecteurs et matrices représentés sous la forme de tableaux Fortran. Les mécanismes de la programmation par objets nous ont fort heureusement permis de faire en sorte que l'interfaçage de Paladin avec le noyau BLAS soit réalisé de manière transparente et n'impose aucune restriction sur le développement ultérieur de la bibliothèque.

4.5.2 Caractérisation des objets compatibles avec BLAS

Pour interfacer Paladin avec le noyau BLAS, nous avons construit deux classes abstraites BLAS_MATRIX et BLAS_VECTOR caractérisant les objets Eiffel dont la représentation interne est « compatible » avec celle des matrices et vecteurs de Fortran, seuls objets que les primitives BLAS puissent accepter comme paramètres.

Cas des matrices

Pour qu'une matrice puisse être passée en paramètre à une primitive BLAS, sa représentation interne doit être conforme à celle des tableaux Fortran, c'est-à-dire que les éléments d'une même colonne doivent être contigus en mémoire. En revanche les colonnes peuvent être disjointes en mémoire, pourvu que deux éléments d'une même ligne soient séparés par un nombre constant de « cases mémoire ». Ce nombre caractérise la « dimension dominante » (*leading dimension*) de la matrice. Une matrice A de taille $m \times n$ devant être passée en paramètre à une primitive BLAS doit donc être décrite par un quintuplet (ptr, m, n, ld), où ptr désigne l'adresse en mémoire du premier élément $A_{1,1}$ de la matrice considérée et ld désigne sa « dimension dominante ».

Nous avons développé la classe BLAS_MATRIX (dont l'interface est reproduite dans l'exemple 4.17) afin qu'elle serve de lien entre le monde des objets matrices de

d'optimisation *-O4*. Les routines BLAS étaient les routines encapsulées dans la bibliothèque *libkmath.a* faisant partie de l'environnement logiciel de la machine Paragon.

Paladin et celui des « objets » pouvant être manipulés par le noyau BLAS. La classe BLAS_MATRIX hérite de la classe abstraite MATRIX : un objet de type BLAS_MATRIX est avant tout une matrice au sens où on l'entend dans Paladin. Cet objet devant aussi pouvoir être passé en paramètre lors de l'invocation d'une routine du noyau BLAS, il est caractérisé non seulement par son type et ses dimensions (attributs *nrow* et *ncolumn* hérités de la classe MATRIX), mais aussi par des informations supplémentaires (les primitives *ld*, *area* et *offset*) permettant de connaître lors de l'invocation d'une routine BLAS la dimension dominante de la matrice considérée et l'adresse de son premier élément.

Exemple 4.17

```
class interface BLAS_MATRIX inherit
    MATRIX
feature {BLAS_MATRIX} -- Compatibility with BLAS
    area : SPECIAL [T]
            -- Special data zone                                    5
    offset : INTEGER
            -- Offset of first element with respect to 'area'
    ld : INTEGER
            -- Number of storage locations between elements in the same row
end -- interface BLAS_MATRIX                                       10
```

La primitive *area* sert à localiser le début de la zone de données d'un objet Eiffel de type tableau. En introduisant en outre la primitive *offset* dans la classe BLAS_MATRIX, on est en mesure de faire en sorte que les sous-matrices issues d'une matrice de type BLAS_MATRIX soient elles-mêmes de type BLAS_MATRIX (la mise en œuvre de la classe SUB_BLAS_MATRIX n'est pas décrite dans ce document, mais la figure 4.6 permet de visualiser comment cette classe combine les caractéristiques des classes SUB_MATRIX et BLAS_MATRIX).

Cas des vecteurs

Nous avons procédé de manière similaire pour caractériser les objets Eiffel ayant une représentation interne compatible avec celle des vecteurs Fortran reconnus par les routines BLAS. Un vecteur V de taille n devant être passé en paramètre à une primitive BLAS doit être décrit par un triplet $(ptr, n, stride)$, où ptr désigne l'adresse en mémoire de l'élément V_1 et $stride$ est le nombre de « cases mémoire » séparant deux éléments successifs du vecteur (un vecteur Fortran peut occuper une zone contiguë ou discontiguë en mémoire, pourvu que ses éléments soient espacés régulièrement).

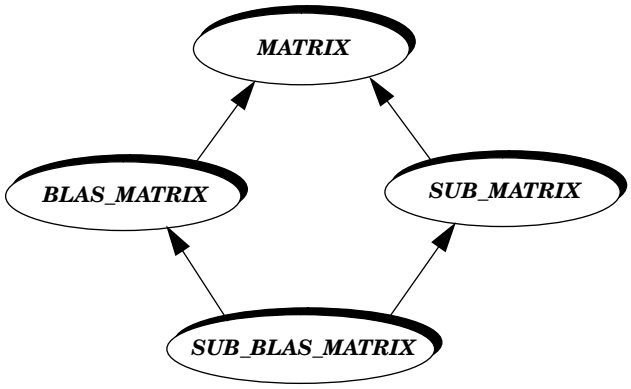


FIG. 4.6 - *Caractérisation des matrices compatibles avec BLAS*

Exemple 4.18

```
class interface BLAS_VECTOR inherit
    VECTOR
feature {BLAS_VECTOR} -- Compatibility with BLAS
    area: SPECIAL [T]
            -- Special data zone                                    5
    offset: INTEGER
            -- Offset of first element with respect to 'area'
    stride: INTEGER
            -- Number of storage locations between elements
end -- interface BLAS_VECTOR                                       10
```

La classe BLAS_VECTOR (dont l'interface est reproduite dans l'exemple 4.18) hérite de la classe abstraite VECTOR et on y a déclaré trois primitives *area*, *offset* et *stride*. Les deux premières primitives jouent le même rôle que leurs homonymes respectifs dans la classe BLAS_MATRIX : elle permettent de localiser le début de la zone de données d'un objet vecteur Eiffel, et l'on peut en outre considérer les sous-vecteurs issus d'un objet de type BLAS_VECTOR comme étant eux mêmes de type BLAS_VECTOR.

4.5.3 Obtention d'objets Eiffel compatibles avec BLAS

Pour que le noyau BLAS puisse être effectivement utilisé lors des calculs réalisés avec Paladin, il a fallu faire en sorte que la représentation interne de certains des

objets manipulés dans Paladin soit compatible avec celle des vecteurs et matrices reconnus par les routines BLAS.

4.5.3.1 Cas des vecteurs locaux

Il nous a suffi de modifier très légèrement la classe LOCAL_VECTOR afin de faire des instances de cette classes des objets de type conforme à BLAS_VECTOR. Dans sa première mise en œuvre, la classe LOCAL_VECTOR héritait directement des classes VECTOR et ARRAY. Pour que les vecteurs locaux de Paladin deviennent des vecteurs compatibles avec BLAS, nous avons simplement modifié la clause d'héritage de la classe LOCAL_VECTOR et défini de manière adéquate les routines *stride* et *offset* héritées de BLAS_VECTOR, comme illustré dans l'exemple 4.19.

Exemple 4.19

```
class LOCAL_VECTOR inherit
    BLAS_VECTOR
    ARRAY [DOUBLE]
    ...
feature -- Compatibility with BLAS                                    5
    stride: INTEGER is 1;
    offset: INTEGER is 0;
end -- LOCAL_VECTOR
```

La classe LOCAL_VECTOR hérite à présent de BLAS_VECTOR, et non plus directement de VECTOR (figure 4.7). Pour les vecteurs locaux les primitives *stride* et *offset* peuvent prendre des valeurs constantes. La primitive *area* déclarée dans la classe BLAS_VECTOR est directement fusionnée dans le cadre de l'héritage multiple avec l'attribut *area* défini dans la classe ARRAY. Cet attribut référence un objet de type SPECIAL [29], qui constitue en Eiffel l'objet dans lequel sont stockées les données d'un tableau.

4.5.3.2 Cas des matrices locales

Dans sa première mise en œuvre, la classe LOCAL_MATRIX héritait directement des classes MATRIX et ARRAY2. Cependant, le format de représentation décrit dans la classe ARRAY2 s'inspire de celui des tableaux bi-dimensionnels dans le langage C : les éléments y sont stockés dans le sens des lignes, alors que les routines BLAS n'admettent que des matrices rangées dans les sens des colonnes. Nous avons

29. La classe SPECIAL est l'une des classes de la bibliothèque Eiffel standard.

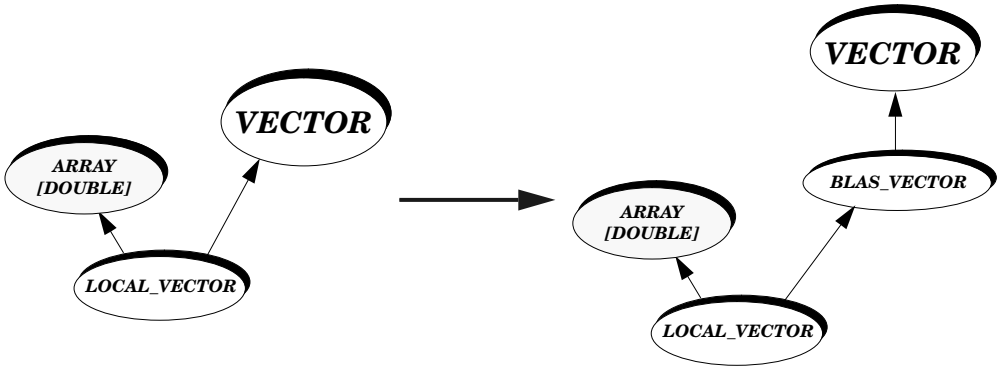


FIG. 4.7 - *Modification de la structure d'héritage afin de faire des vecteurs locaux des objets compatibles avec BLAS*

donc dû développer une classe MY_ARRAY2 présentant la même interface que la classe standard ARRAY2, mais décrivant un format de représentation des tableaux compatible avec celui des tableaux Fortran. Il nous a ensuite suffi de modifier la clause d'héritage de la classe LOCAL_MATRIX afin de la faire hériter de BLAS_MATRIX et de MY_ARRAY2 (figure 4.8), et de définir de manière adéquate les primitives *ld* et *offset* (la primitive *area* étant directement fusionnée avec l'attribut *area* de la classe MY_ARRAY2).

Exemple 4.20

```
class LOCAL_MATRIX inherit
    BLAS_MATRIX
    MY_ARRAY2 [DOUBLE]
    ...
feature -- Compatibility with BLAS                                    5
    ld: INTEGER is do Result := nrow end;
    offset: INTEGER is 0;
end -- LOCAL_MATRIX
```

4.5.3.3 Autres types d'objets compatibles avec BLAS

Outre les deux types de base LOCAL_VECTOR et LOCAL_MATRIX, nous avons considéré plusieurs autres types d'objets pouvant être perçus comme étant compatibles avec BLAS.

Ainsi, pour tout vecteur colonne extrait d'une matrice de type BLAS_MATRIX on peut assez aisément calculer le triplet caractéristique $(ptr, size, stride)$ en s'ap-

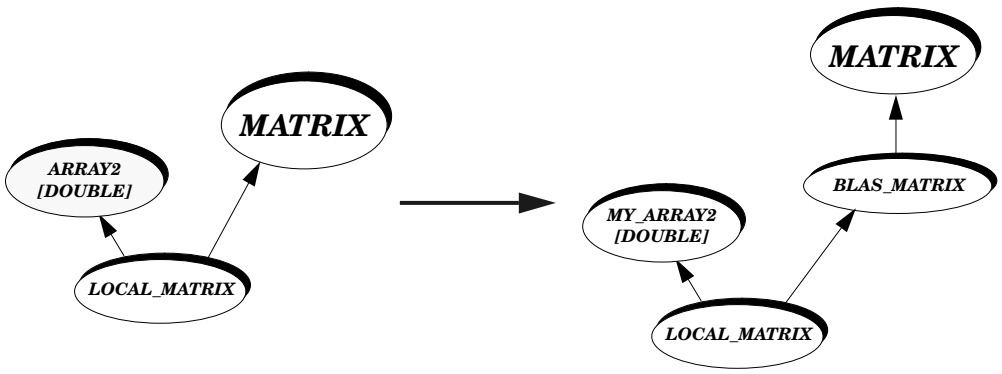


FIG. 4.8 - *Modification de la structure d'héritage afin de faire des matrices locales des objets compatibles avec BLAS*

puyant sur le numéro de la colonne considérée et sur les informations caractérisant la matrice englobante. Un objet de ce type est donc conforme à BLAS_VECTOR. Nous avons enrichi la hiérarchie des classes de Paladin en conséquence : la classe BLAS_COLUMN caractérise les vecteurs définis comme étant une vue (BLAS_COLUMN hérite de COLUMN) sur une colonne d'une matrice de type BLAS_MATRIX, ces vecteurs ayant un format de représentation interne compatible avec celui des vecteurs reconnus par les routines BLAS (BLAS_COLUMN hérite aussi de BLAS_VECTOR).

Partant des types BLAS_VECTOR et BLAS_MATRIX, nous avons ainsi étendu la hiérarchie des classes de Paladin. Les classes BLAS_COLUMN, BLAS_ROW, BLAS_DIAGONAL, SUB_BLAS_VECTOR, et SUB_BLAS_MATRIX caractérisent toutes des vecteurs et matrices pouvant être passés en paramètres à des routines BLAS (voir figure 4.9).

4.5.4 Interface avec le noyau BLAS

La classe BLAS constitue l'interface entre le monde des routines Eiffel et le noyau BLAS. On a reproduit dans l'exemple 4.21 une partie de l'interface de cette classe.

La classe BLAS peut en fait être perçue comme offrant au noyau BLAS une interface conforme aux principes de typage et de masquage d'information de la programmation par objets. Les routines de cette classe prennent directement en paramètres des objets de type BLAS_VECTOR ou BLAS_MATRIX. Elles sont en outre dotées de préconditions qui contribuent à rendre plus sûr l'emploi du noyau BLAS dans le monde Eiffel.

Du point de vue d'un programmeur développant des classes Eiffel et utilisant les routines de la classe BLAS, celle-ci offre une abstraction algorithmique : elle permet au programmeur de bénéficier du savoir-faire encapsulé dans le noyau BLAS par ses

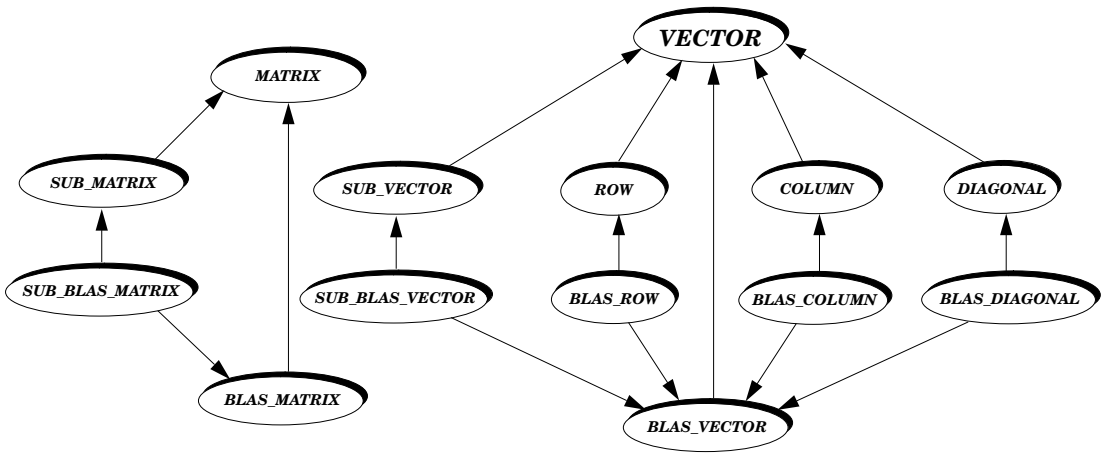


FIG. 4.9 - *Caractérisation des objets compatibles avec BLAS*

concepteurs, sans avoir à connaître les détails de mise en œuvre de ce noyau.

Nous n'avons pu faire en sorte d'appeler les routines BLAS directement depuis le corps des routines de la classe BLAS. La raison en est que, avant de pouvoir passer en paramètre à une routine BLAS l'adresse du premier élément d'un objet vecteur ou d'un objet matrice, il nous faut d'abord *calculer* cette adresse. Or, avec l'environnement Eiffel d'ISE que nous avons utilisé pour développer et expérimenter Paladin, les objets Eiffel sont susceptibles d'être déplacés à tout instant en mémoire à l'initiative du ramasse-miettes, lequel a la charge de gérer la mémoire en lui évitant notamment d'être trop fragmentée. En calculant l'adresse d'un objet directement dans le corps d'une routine Eiffel avant de la passer en paramètre à une routine externe, on court donc le risque de voir cette adresse invalidée entre la fin du calcul d'adresse et l'appel de la routine externe.

Nous avons donc dû développer un petit module C jouant le rôle d'intermédiaire entre la classe BLAS et le noyau BLAS. Ce petit module, baptisé *eif_to_blas* (voir figure 4.10), contient des fonctions C au niveau desquelles on procède au calcul d'adresse qui n'a pu être réalisé dans la classe BLAS. Les routines de la classe BLAS ne sont donc pas définies directement dans cette classe. Elles sont simplement déclarées comme étant des routines *externes*. Ainsi, la routine *scal* (qui réalise un produit scalaire-vecteur) est simplement déclarée comme illustré dans l'exemple 4.22.

Cette déclaration indique au compilateur Eiffel qu'il existe une fonction C externe baptisée *scal*, et que c'est cette fonction qui doit être exécutée lorsque la routine *scal* de la classe BLAS est invoquée dans le cadre d'une application Eiffel.

La fonction C *scal* est donc implantée dans le module *eif_to_blas* de manière à calculer l'adresse du segment de données de l'objet de type BLAS_VECTOR passé

Exemple 4.21

```
expanded class interface BLAS

feature -- Level 1 BLAS
   scal (alpha: DOUBLE; X: BLAS_VECTOR) is
         -- General scalar-vector product                              5
         -- X <- alpha * X
   ...
feature -- Level 2 BLAS
   gemv (alpha: DOUBLE; transA: CHARACTER; A: BLAS_MATRIX;
          X: BLAS_VECTOR; beta: DOUBLE; Y: BLAS_VECTOR) is            10
         -- General matrix-vector product
         -- Y <- alpha * A^(transA) * X + beta * Y
      require
         size_ok: (X.length = A.ncolumn) and (A.nrow = Y.length)
   ...                                                                 15
feature -- Level 3 BLAS
   gemm (alpha: DOUBLE; transA: CHARACTER; A: BLAS_MATRIX;
         transB: CHARACTER; B: BLAS_MATRIX;
         beta: DOUBLE; C: BLAS_MATRIX) is
         -- General matrix-matrix product                             20
         -- C <- alpha * A^(transA) * B^(transB) + beta * C
      require
         size_ok: (C.nrow = A.nrow) and (C.ncolumn = B.ncolumn)
                  and (A.ncolumn = B.nrow)
   ...                                                                 25
end -- interface BLAS
```

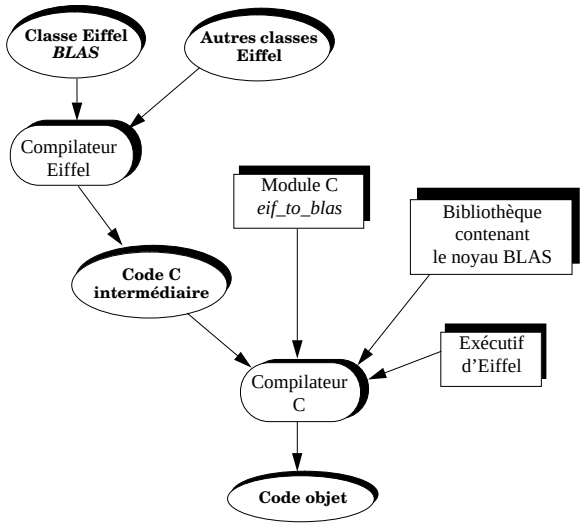


FIG. 4.10 - *Interfaçage d'Eiffel avec le noyau BLAS*

Exemple 4.22

```
expanded class BLAS

feature -- Level 1 BLAS
   scal (alpha: DOUBLE; X: BLAS_VECTOR) is
         -- General scalar-vector product                    5
         -- X <- alpha * X
      external "C" end; -- scal
   ...
end -- class BLAS
```

en paramètre, et à invoquer ensuite la routine `DSCAL` du noyau BLAS avec les paramètres appropriés.

La bibliothèque Cecil (*C Eiffel Call-In Library*) associée au langage Eiffel fournit toutes les primitives nécessaires pour accéder à des objets Eiffel depuis des fonctions C [30]. En utilisant certaines de ces primitives, on peut notamment calculer au niveau du module *eif_to_blas* l'adresse des segments de données des objets vecteurs et matrices passés en paramètres aux routines de la classe BLAS.

30. Pour plus de détails sur la bibliothèque Cecil, voir par exemple le §27.4 de [97].

Retour sur la généricité

La version actuelle de Paladin n'étant pas générique mais seulement dédiée aux calculs impliquant des vecteurs et matrices de nombres réels représentés en double précision, la classe BLAS ne permet pour l'instant que l'interfaçage avec les routines correspondantes du noyau BLAS (*e.g.* `DSCAL`, `DDOT`, `DGEMM`). Il serait cependant possible de rendre la classe BLAS générique et, en testant le type des objets passés en paramètres, de réaliser un branchement vers les routines de BLAS adéquates selon qu'on a à manipuler des vecteurs et matrices à valeurs entières, à valeurs réelles (en simple ou en double précision), à valeurs complexes, etc.

4.5.5 Techniques d'optimisation associées

Les classes BLAS_VECTOR et BLAS_MATRIX héritent des nombreux opérateurs définis dans les classes abstraites VECTOR et MATRIX. Pour que l'utilisateur ne soit pas contraint d'invoquer explicitement les routines de la classe BLAS, nous avons redéfini dans les classes BLAS_MATRIX et BLAS_VECTOR certains opérateurs afin que leur invocation entraîne automatiquement l'appel de la routine BLAS correspondante, lorsqu'une telle routine existe.

Opérateurs unaires

Considérons la routine *scal* définie dans la classe VECTOR. Dans cette classe la routine *scal* est dotée d'une mise en œuvre séquentielle par défaut basée sur une simple itération au cours de laquelle chaque élément du vecteur courant est multiplié par la valeur scalaire passée en paramètre.

Dans la classe BLAS_VECTOR, nous avons encapsulé un objet *agent*[31] de type BLAS, que nous avons ensuite utilisé pour redéfinir la routine *scal* comme montré dans l'exemple 4.23.

Le mécanisme de la liaison dynamique assure une transparence totale pour l'utilisateur. Chaque fois que la routine *scal* sera invoquée sur un objet d'un type conforme à BLAS_VECTOR (par exemple sur un vecteur local), la routine *scal* de la classe BLAS et par conséquent la routine `DSCAL` du noyau BLAS seront automatiquement invoquées.

En redéfinissant ainsi dans les classes BLAS_VECTOR et BLAS_MATRIX tous les opérateurs unaires (du moins ceux pour lesquels il existe une routine équivalente dans le noyau BLAS), on assure donc que les routines BLAS seront invoquées systématiquement — et de manière totalement transparente pour l'utilisateur — chaque fois que la représentation interne de l'objet courant s'y prêtera.

31. Les objets *agents* ont été introduits au § 3.2.4.

Exemple 4.23

```
deferred class BLAS_VECTOR inherit
    VECTOR
        redefine scal end
feature {NONE} -- Interface with BLAS kernel
    BLAS: BLAS;                                                    5
feature -- Optimized operators
    scal (alpha: like item) is
        do
            BLAS.scal (alpha, Current);
        end;                                                       10
    ...
end -- BLAS_VECTOR
```

Opérateurs N-aires

Le problème est un peu plus complexe lorsque l'opérateur considéré implique plusieurs paramètres pouvant tous être compatibles ou non avec BLAS. Il faut alors tester le type de chaque objet passé en paramètre afin de vérifier s'il est conforme à BLAS_VECTOR ou à BLAS_MATRIX. Ce problème se ramène à un problème de mise en œuvre manuelle d'un mécanisme de sélection dynamique multiple. On peut donc appliquer les techniques décrites au paragraphe 4.4.

On a reproduit dans l'exemple 4.24 la mise en œuvre de la fonction *dot* dans la classe BLAS_VECTOR. Cette fonction calcule le produit scalaire de deux vecteurs et peut dans ce but faire éventuellement appel à la routine `DDOT` du noyau BLAS.

Chaque fois que les deux vecteurs impliqués dans un calcul de produit scalaire seront d'un type conforme à BLAS_VECTOR, la routine `DDOT` du noyau BLAS sera automatiquement invoquée (via la routine `dot` de la classe BLAS). Dans le cas contraire, l'algorithme par défaut hérité de la classe MATRIX sera exécuté.

Redéfinition des accesseurs

Nous avons redéfini dans les classes BLAS_VECTOR et BLAS_MATRIX certains accesseurs de manière à ce qu'ils retournent des objets compatibles avec BLAS chaque fois que c'est possible. Ainsi, alors que l'accesseur *row* est défini dans la classe MATRIX de manière à retourner un objet de type ROW, cet accesseur a été redéfini dans la classe BLAS_MATRIX afin de retourner un objet de type BLAS_ROW. Pour l'utilisateur, le type exact de l'objet retourné importe peu : il lui suffit de savoir que cet objet peut être manipulé *comme n'importe quel autre vecteur*. Par contre, la redéfinition de l'accesseur *row* dans BLAS_MATRIX permet d'assurer que dans tout

Exemple 4.24

```
deferred class BLAS_VECTOR
inherit
    VECTOR
        rename dot as dot_default end
    ...                                                          5
feature -- Optimized operators
    dot (x: VECTOR): like item is
        local
            tmp_X: BLAS_VECTOR;
        do                                                       10
            tmp_X?= x;    -- Assignment attempt

            if (tmp_X /= Void) then
                Result := BLAS.dot (tmp_X, Current);
            else                                                 15
                Result := dot_default (x);
            end; -- if
        end;
    ...
end -- BLAS_VECTOR                                               20
```

algorithme réalisant des opérations portant sur les lignes d'une matrice de type conforme à BLAS_MATRIX (par exemple une matrice locale), les routines du noyau BLAS seront invoquées partout où les calculs le permettront.

Vision du programmeur d'application

Dans l'exemple 4.25, on illustre le fait que l'interfaçage de Paladin avec le noyau BLAS est maintenu totalement transparent pour l'utilisateur, les routines BLAS étant automatiquement invoquées pour effectuer les calculs lorsque le format des opérandes s'y prête.

Exemple 4.25

```
local
    A : LOCAL_MATRIX;
    my_row, my_col : VECTOR;
    v : DOUBLE;
do                                                    5
   !!A.make (10, 10);
    my_row := A.row (6);
    my_col := A.col (3);
    ...
    my_row.scal (3.14);                              10
    v := my_row.dot (my_col);
    ...
end;
```

- On crée tout d'abord une matrice A de taille 10×10 et de type LOCAL_MATRIX. La matrice référencée par A est donc d'un type conforme à BLAS_MATRIX, ce qui revient à dire que sa représentation interne est conforme à celle exigée par les routines du noyau BLAS.

- On invoque ensuite les accesseurs *row* et *column* sur la matrice A. Les objets retournés par ces accesseurs sont de type BLAS_ROW et BLAS_COLUMN respectivement (grâce à la redéfinition de ces accesseurs dans BLAS_MATRIX), mais le programmeur d'application n'a pas à s'en soucier : il lui suffit de savoir qu'il s'agit d'objets conformes au type VECTOR.

- Lors de l'invocation de la routine *scal* sur l'objet référencé par *my_row*, c'est en fait la routine `DSCAL` du noyau BLAS qui va automatiquement être appelée

grâce à la liaison dynamique (le type dynamique de la variable *my_row* étant conforme à BLAS_VECTOR) pour effectuer l'opération désirée.

- De même, lorsque la routine *dot* est invoquée sur *my_row* avec en paramètre *my_column*, le mécanisme de sélection dynamique multiple implanté dans la classe BLAS_VECTOR assure que la routine `DDOT` de BLAS va être appelée pour calculer le produit scalaire des deux vecteurs.

4.5.6 Fonctionnement de Paladin sans le noyau BLAS

L'interfaçage de Paladin avec le noyau BLAS a été réalisé de telle sorte que dans sa version actuelle, Paladin ne peut plus fonctionner sans utiliser BLAS. Pourtant, pour que la portabilité de Paladin ne soit pas compromise, il faudrait que l'utilisation de BLAS demeure complètement optionnelle, ne serait-ce que parce que sur certaines plate-formes parallèles le noyau BLAS peut ne pas être disponible, ou parce qu'on peut préférer ne pas l'utiliser sur une plate-forme donnée.

On peut envisager au moins deux façons de procéder pour rendre la bibliothèque Paladin indépendante de la disponibilité du noyau BLAS. La première approche serait de tester la disponibilité du noyau BLAS avant chaque invocation possible d'une routine de la classe BLAS. Ainsi, la définition de la routine *scal* dans la classe BLAS_VECTOR (déjà évoquée au § 4.5.5) deviendrait telle que reproduite dans l'exemple 4.26.

La deuxième approche consisterait à maintenir deux classes descendant de la classe BLAS, la première faisant effectivement appel aux routines routines externes du noyau BLAS alors que dans la seconde tous les algorithmes de calcul seraient exprimés en Eiffel. On pourrait par exemple baptiser ces deux classes REAL_BLAS et EIFFEL_BLAS respectivement. Le choix de l'une ou l'autre classe pourrait être effectué, soit statiquement lors de la compilation (le langage *Lace*[32] qui sert à décrire un système de classes Eiffel permet de procéder à des renommages et à des substitutions de classes), soit dynamiquement en intégrant dans les classes BLAS_VECTOR et BLAS_MATRIX une définition telle que celle reproduite dans l'exemple 4.27.

Dans cet exemple, on crée un objet agent unique (voir le point de langage 4.1 à la page 160) de type BLAS qui va ensuite servir de « pointeur » vers l'ensemble des routines de calcul fournies par la classe choisie. Selon que l'on crée en pratique un objet de type REAL_BLAS ou de type EIFFEL_BLAS, l'invocation d'une routine sur cet objet se traduira soit par l'appel de la routine BLAS correspondante, soit par l'exécution d'un algorithme équivalent exprimé en Eiffel.

32. *Lace : Language for Assembly of Classes in Eiffel* [98].

Exemple 4.26

```
deferred class BLAS_VECTOR inherit
    VECTOR
        rename scal as scal_default end

feature -- Optimized operators                                          5
    scal (alpha : like item) is
        do
            if (Use_Real_Blas) then
                BLAS.scal (alpha, Current);
            else                                                        10
                scal_default (alpha);
            end; -- if
        end;
    ...
end -- BLAS_VECTOR                                                      15
```

Exemple 4.27

```
deferred class BLAS_VECTOR inherit
    ...
feature {NONE} -- Interface with BLAS kernel
     BLAS : BLAS is
        once                                                            5
            if (Use_Real_Blas) then
               !REAL_BLAS !Result;
            else
               !EIFFEL_BLAS !Result;
            end; -- if                                                  10
        end; -- blas
    ...
end -- BLAS_VECTOR
```

Point de langage 4.1

Le mot-clé once. Le mot-clé **once** placé au début du corps d'une routine définie dans une classe C quelconque signifie que le code de cette routine ne doit être exécuté qu'une seule et unique fois lors de la première invocation de cette routine (pour toutes les instances de la classe C et de ses descendantes). Lorsque cette routine est une fonction, tous les appels subséquents retourneront directement le résultat retourné lors de la première invocation de cette fonction. Ce mécanisme permet notamment à tous les objets d'un même type de référencer un objet commun.

Chapitre 5

Mise en œuvre du polymorphisme

Il peut dans certains cas être intéressant d'adapter dynamiquement le format de représentation interne d'un agrégat aux exigences du calcul en cours. Une matrice distribuée par blocs, par exemple, peut devoir être redistribuée dynamiquement. Il peut aussi s'avérer souhaitable de transformer une matrice locale en une matrice distribuée, parce que cette nouvelle représentation est supposée pouvoir amener à de meilleurs performances. Il y a cependant une différence entre ces deux types de conversion. Lorsque l'on modifie le schéma de distribution d'une matrice distribuée par blocs, le type de l'objet considéré n'a pas à être modifié : cet objet demeure une instance de la classe DBLOCK_MATRIX. En revanche, transformer une matrice locale en une matrice distribuée par blocs nécessite que le type de l'objet soit modifié : la matrice doit abandonner le type LOCAL_MATRIX pour adopter le nouveau type DBLOCK_MATRIX.

Les techniques permettant de changer la représentation interne d'un agrégat polymorphe varient selon que le type de l'agrégat doit être modifié ou non au cours de l'opération. Dans le paragraphe 5.1, on décrit les techniques utilisées dans Paladin pour procéder à la redistribution des matrices. En général, ces techniques n'impliquent pas de changement du type d'une matrice lors de sa redistribution. On aborde ensuite dans le paragraphe 5.2 les mécanismes permettant de changer le type d'une matrice. On détaille tout d'abord la procédure utilisée dans la version actuelle de Paladin, en montrant que cette procédure n'est pas totalement satisfaisante. On décrit ensuite une méthode alternative qui devrait permettre la mise en œuvre de matrices réellement polymorphes dans Paladin.

5.1 Redistribution des matrices

5.1.1 Cas des matrices distribuées par blocs

Des algorithmes permettant de redistribuer une matrice de type DBLOCK_MATRIX peuvent être développés en s'appuyant sur les mécanismes fournis par les classes POM et TRANSMISSIBLE et sur les mécanismes de gestion de la distribution fournis par la classe DISTRIBUTION_2D (et ses descendantes). Ces algorithmes se distinguent par le type de redistribution considéré. La redistribution est en effet plus ou moins difficile à réaliser selon que les schémas de distribution source et cible sont très dissemblables ou non. À titre d'exemple, deux des routines de redistribution que nous avons implantées dans la classe DBLOCK_MATRIX sont décrites dans les paragraphes qui suivent.

5.1.1.1 Redistribution sans changement du partitionnement

La routine *change_mapping* reproduite dans l'exemple 5.1 est dédiée à la redistribution d'une matrice lorsque cette redistribution n'implique pas de changement du partitionnement. Ni la taille des blocs de partitionnement, ni les dimensions de la table des blocs ne sont donc altérés au cours de la redistribution. Seul le placement des blocs peut être affecté par l'exécution de cette routine.

- La routine *change_mapping* admet en paramètre un descripteur de distribution baptisé *new_dist*, décrivant le nouveau schéma de distribution devant être adopté par la matrice courante.

- La précondition *same_partition* en lignes 8 et 9 impose que les facteurs de blocs soient identiques dans les schémas de distribution source (celui de la matrice courante au début de la redistribution) et cible (celui décrit par *new_dist*). Seul le placement peut donc différer entre ces deux schémas.

- Le partitionnement étant conservé dans le type de redistribution considéré ici, il n'est pas nécessaire de créer une nouvelle table des blocs. La redistribution – que l'on devrait plutôt qualifier ici de simple *replacement* — consiste à faire migrer les blocs entre les nœuds (en mettant à jour la table des blocs en conséquence sur chaque nœud) jusqu'à obtenir un placement des blocs conforme au schéma décrit par le paramètre *new_dist*.

- Pour chaque bloc (bi, bj) de la table des blocs ;

  - on teste l'identité du nœud propriétaire de ce bloc dans les schémas de distribution source et cible, et si le bloc doit changer de propriétaire

Exemple 5.1

```
class DBLOCK_MATRIX inherit
    ...
feature -- Matrix redistribution
    change_mapping (new_dist: DISTRIBUTION_2D) is
        require                                                          5
            same_size: (dist.nrow = new_dist.nrow)
                        and (dist.ncolumn = new_dist.ncolumn);
            same_partition: (dist.bfi = new_dist.bfi)
                            and (dist.bfj = new_dist.bfj);
        local                                                            10
            bi, bj, source, target: INTEGER;
            new_block: like local_block;
        do
            from bi:= 0 until bi > dist.nbimax loop
                from bj:= 0 until bj > dist.nbjmax loop                  15
                    source:= owner_of_block (bi, bj);
                    target:= new_dist.owner_of_block (bi, bj);
                    if (source /= target) then
                        new_block:= bring_block_to (bi, bj, target);
                        if (source = POM.node_id) then                   20
                            -- Dispose of local block (GC)
                            put_block (Void, bi, bj);
                        end; -- if
                        if (target = POM.node_id) then
                            -- Put block in table of blocks             25
                            put_block (new_block, bi, bj);
                        end; -- if
                    end; -- if
                    bj:= bj + 1;
                end; -- loop                                             30
                bi:= bi + 1
            end; -- loop
            dist:= new_dist;
        end; -- change_mapping
    ...                                                                  35
end -- DBLOCK_MATRIX
```

on invoque la fonction *bring_block_to* décrite au paragraphe 3.4.1.4 pour effectuer le transfert requis (ligne 19) ;

- on déréférence alors ce bloc dans la table des blocs du nœud source (ligne 22) afin qu'il puisse être récupéré par le ramasse-miettes local ;
- on procède de manière inverse au niveau du nœud destinataire en plaçant le bloc reçu dans la table des blocs locale (ligne 26).

- Lorsque le transfert des blocs est terminé, on affecte le nouveau descripteur de distribution *new_dist* à l'attribut *dist* de la matrice courante (ligne 32). L'ancien descripteur de distribution n'étant plus référencé par la matrice courante, il peut alors être récupéré par le ramasse-miettes (à moins qu'il ne soit encore référencé par une autre matrice distribuée).

- Lorsque l'exécution de la routine *change_mapping* arrive à son terme, la distribution de la matrice courante est conforme au schéma décrit par le descripteur *new_dist*.

5.1.1.2 Redistribution avec changement du partitionnement

La routine *change_mapping* détaillée ci-dessus permet de redistribuer une matrice de manière relativement efficace lorsque seul le placement des blocs doit être modifié. La routine *change_distribution* reproduite dans l'exemple 5.2 est beaucoup plus polyvalente : avec elle une matrice peut être redistribuée même si les paramètres de partitionnement — et par conséquent les dimensions de la table des blocs et celles des blocs eux-mêmes — doivent changer au cours de la redistribution. Ceci impose qu'une nouvelle table des blocs soit créée avec les dimensions requises par le nouveau schéma de distribution, et que les données soient ensuite transférées des anciens blocs vers les nouveaux blocs en tenant compte des recouvrements possibles : un bloc du schéma de distribution source peut couvrir plusieurs blocs du schéma cible, et peut donc avoir à être transmis à plusieurs nœuds destinataires).

Le code de la routine *change_distribution* est reproduit dans l'exemple 5.2 et détaillé ci-dessous.

- Tout comme la routine *change_mapping*, la routine *change_distribution* prend en paramètre un descripteur de distribution baptisé *new_dist*, décrivant le nouveau schéma de distribution devant être adopté par la matrice courante. Contrairement à *change_mapping*, cependant, elle n'impose aucune contrainte sur les paramètres de partitionnement dans les schémas de distribution source et cible. La précondition en lignes 6 et 7 impose simplement que les deux descripteurs de distribution concernent un domaine d'indices de mêmes dimensions.

- La table des blocs de la matrice courante devant être remplacée par une nouvelle table au cours de la redistribution, on crée en fait une matrice temporaire *new_mat* de type DBLOCK_MATRIX à laquelle on passe en paramètre le descripteur de distribution *new_dist* (ligne 15) décrivant le schéma de distribution cible.

- Pour chaque bloc (bi, bj) de la matrice courante :

  - on évalue la section rectangulaire recouverte par ce bloc (lignes 20 et 21) ;
  - on calcule ensuite l'ensemble des nœuds propriétaires des blocs de la matrice *new_mat* recouvrant cette section[1] (ligne 24) ;
  - on utilise la fonction *bring_block_to_many*[2] pour créer et rafraîchir sur chacun des nœuds cibles une copie du bloc (bi, bj) (ligne 26) ;
  - sitôt après que le bloc (bi, bj) a été émis vers tous les nœuds destinataires, on déréférence ce bloc au niveau de la table des blocs de la matrice courante afin qu'il puisse être récupéré par le ramasse-miettes (ligne 28) ;
  - on utilise le mécanisme des vues pour désigner sur la matrice *new_mat* la même section rectangulaire que celle couverte par le bloc (bi, bj) dans la matrice courante, et l'on transfère dans cette section l'information reçue[3] (ligne 30). La routine *convert* invoquée pour effectuer le transfert des données est décrite au paragraphe 5.2.2.

- Lorsque le transfert des blocs est terminé, on affecte le descripteur de distribution *dist* de la matrice *new_mat* à l'attribut *dist* de la matrice courante (ligne 35). On affecte de même l'attribut *area* de la matrice *new_mat*[4] à l'attribut *area* de la matrice courante. Celle-ci abandonne donc son ancienne table des blocs pour adopter celle de la matrice *new_mat*.

- Lorsque l'exécution de la routine *change_distribution* arrive à son terme, la matrice courante est dotée d'un nouveau descripteur de distribution et d'une nouvelle table des blocs conformes aux contraintes de distribution décrites par le paramètre *new_dist*. L'ancien descripteur de distribution, l'ancienne

1. La fonction *owners_of_section* est l'une des nombreuses fonctions de localisation des données offertes par la classe DISTRIBUTION_2D.

2. La fonction *bring_block_to_many* est semblable à la fonction *bring_block_to* décrite au paragraphe 3.4.1.4, mais elle prend en paramètre un ensemble de destinataires au lieu d'un seul.

3. Le mécanisme des vues n'est pas affecté par la distribution éventuelle des objets référencés. L'écriture dans une vue respecte donc la règle des écritures locales, chaque nœud ne mettant à jour que les données dont il est propriétaire.

4. L'attribut *area* référence dans la classe DBLOCK_MATRIX la table des blocs de la matrice courante. Il s'agit de l'un des attributs hérités de la classe ARRAY2.

table des blocs, et la matrice *new_mat* elle-même peuvent être récupérés par le ramasse-miettes [5].

Discussion

En développant l'algorithme encapsulé dans la routine *change_distribution*, nous avons *choisi* d'organiser toutes les communications en fonction des blocs de partitionnement du schéma de distribution source. En d'autres termes, chacun des blocs de la matrice courante est expédié dans son intégralité vers un ou plusieurs nœuds destinataires, à charge pour chacun des destinataires de ne retenir alors que la partie de ce bloc qui l'intéresse pour mettre à jour les blocs dont il est propriétaire dans le schéma de distribution cible. Une approche alternative aurait consisté à calculer d'abord au niveau du nœud émetteur, pour chacun des nœuds destinataires, la partie du bloc local intéressant ce destinataire et à ne lui expédier que cette partie. Cette approche aurait à coup sûr été plus économique du point de vue du volume des données transmises, mais elle nous aurait obligés à appliquer des techniques complexes et coûteuses d'agrégation des données au niveau du nœud émetteur, qui ne sont pas toujours rentables sur les plates-formes parallèles [6]. En outre, en procédant de la sorte nous aurions diminué la concurrence possible lors de la redistribution : le nœud émetteur aurait dû calculer *séquentiellement* les données devant être expédiées à chacun des nœuds destinataires, alors qu'avec l'approche retenue ici les nœuds destinataires peuvent procéder *concurremment* à l'extraction des données qui les intéressent à partir du bloc reçu.

Illustration

On a reproduit dans la figure 5.1 un exemple d'utilisation de la routine de redistribution *change_distribution*. Dans cet exemple, il s'agit de transformer une matrice partitionnée en neuf blocs élémentaires en une matrice partitionnée en seulement quatre blocs élémentaires. Pour simplifier, on a noté A la matrice source et B la matrice cible (s'agissant d'une redistribution, il n'y a en réalité qu'une seule matrice impliquée dans l'opération). On a fait apparaître dans la figure 5.1 l'identité du nœud propriétaire de chaque bloc dans les schémas de distribution source et cible. On a également reproduit dans cette figure le diagramme de Hasse exprimant les dépendances causales entre les émissions et réceptions de blocs.

5. L'ancien descripteur de distribution ne sera toutefois récupéré par le ramasse-miettes que s'il n'est plus référencé par aucune autre matrice distribuée.

6. Sur la Paragon XP/S, les copies de mémoire à mémoire sur un même nœud peuvent être plus coûteuses que les communications.

Exemple 5.2

```
class DBLOCK_MATRIX inherit
    ...
feature -- Matrix redistribution
    change_distribution (new_dist: DISTRIBUTION_2D) is
        require
            same_size: (dist.nrow = new_dist.nrow)
                        and (dist.ncolumn = new_dist.ncolumn);
        local
            bi, bj, i1, i2, j1, j2: INTEGER;
            targets: SET [INTEGER];
            copy_of_block: like local_block;
            new_mat: like Current;
        do
            -- Create new matrix with distribution as required
            !!new_mat.make_from (new_dist);
            from bi := 0 until bi > dist.nbimax loop
                from bj := 0 until bj > dist.nbjmax loop
                    -- Compute the index domain (i1, j1)..(i2, j2) covered
                    -- by Current's block (bi, bj)
                    i1 := dist.imin (bi); i2 := dist.imax (bi);
                    j1 := dist.jmin (bj); j2 := dist.jmax (bj);
                    -- Compute the targets, i.e. the owners of the blocks
                    -- that cover (i1, j1)..(i2, j2) in 'new_mat'
                    targets := new_mat.dist.owners_of_section (i1, j1, i2, j2);
                    -- Create a copy of Current's block (bi, bj) on all 'targets'
                    copy_of_block := bring_block_to_many (bi, bj, targets);
                    -- Dispose of local block (GC)
                    put_local_block (Void, bi, bj);
                    -- Update section (i1, j1)..(i2, j2) of 'new_mat'
                    new_mat.submatrix (i1, j1, i2, j2).convert (copy_of_block);
                    bj := bj + 1;
                end; -- loop
                bi := bi + 1
            end; -- loop
            dist := new_mat.dist; area := new_mat.area;
        end; -- change_distribution
    ...
end -- DBLOCK_MATRIX
```

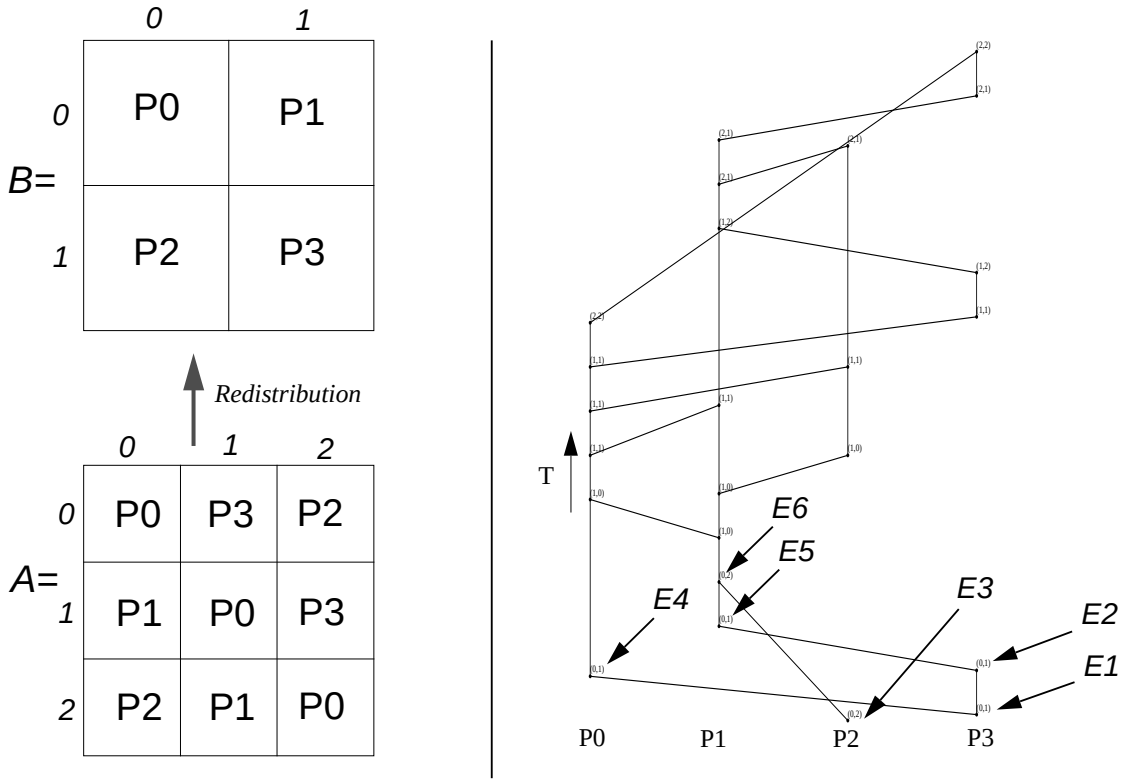


FIG. 5.1 - *Exemple de redistribution réalisée avec la routine* change_distribution

On a détaillé ci-dessous les premiers échanges de blocs réalisés lors de l'exécution de la routine *change_distribution*.

- Le bloc $A_{0,0}$ ayant le même propriétaire que le bloc $B_{0,0}$, ce bloc ne requiert aucune transmission.

- Le bloc $A_{0,1}$ appartenant au nœud $P3$ doit être transmis à la fois aux nœuds $P0$ et $P1$, propriétaires des blocs $B_{0,0}$ et $B_{0,1}$. On peut observer dans le diagramme des transmissions que le nœud noté $P3$ adresse bien deux messages successifs, le premier au nœud $P0$ (événement noté *E1*) et le second au nœud $P1$ (événement *E2*).

- Pendant ce temps, le nœud $P2$, qui n'est concerné par aucun des blocs $A_{0,0}$, $A_{0,1}$, $B_{0,0}$ et $B_{0,1}$, et n'a donc rien eu à émettre ni à recevoir jusqu'à présent, adresse le bloc $A_{0,2}$ dont il est propriétaire au nœud $P1$, propriétaire du bloc $B_{0,1}$ (événement *E3*). La réception de ce bloc (événement *E6*) ne pourra toutefois avoir lieu au niveau du nœud $P1$ qu'après que ce nœud ait reçu le bloc

provenant de $P3$ (événement *E5*). Cette contrainte est due au fait que, telle qu'est elle est implantée actuellement, la routine *change_distribution* s'appuie sur un séquencement strict des émissions et réceptions de blocs sur chaque nœud, les primitives de réception étant en outre des primitives bloquantes. Ce problème fait l'objet du paragraphe 5.1.1.3.

Le diagramme des dépendances causales reproduit dans la figure 5.1 a été obtenu en utilisant les mécanismes d'observation offerts par la bibliothèque POM : on a exécuté la routine *change_distribution* sur une machine offrant 4 processeurs afin de redistribuer une matrice conformément aux schémas de distribution source et cible visualisés dans la figure 5.1. Au cours de cette exécution, les services de génération automatique de trace de la POM ont été activés afin qu'un message de trace portant une estampille vectorielle soit généré lors de chaque émission ou réception d'un bloc pendant la redistribution. Les messages de trace ont été collectés et traités par un programme observateur fonctionnant en parallèle avec l'application. Les informations produites par l'observateur ont ensuite été fournies à un outil de visualisation graphique capable de dessiner le diagramme des dépendances causales à partir d'une liste d'événements estampillés. Le programme observateur et l'outil de visalisation graphique ont été développés par C. Bareau dans le cadre de son travail de thèse [12].

5.1.1.3 Optimisation envisageable

Les algorithmes des deux routines *change_distribution* et *change_mapping* s'articulent autour de deux boucles imbriquées dans lesquelles on procède à l'énumération des blocs de la matrice courante. Les données transférées entre les nœuds sont également des blocs de la matrice courante, les émissions et réceptions étant encapsulées dans les routines *bring_block_to* (détaillée au paragraphe 3.4.1.4) et *bring_block_to_many*.

Dans ces deux routines, on s'appuie sur les primitives de réception bloquantes de la classe POM [7] pour assurer la réception des données au niveau du ou des nœuds destinataires. Lorsqu'un nœud destinataire d'un bloc (i, j) donné invoque la fonction *bring_block_to* au cours de l'exécution de la routine *change_mapping*, il peut se trouver bloqué en attente de ce bloc, alors qu'un autre nœud est lui-même en attente du bloc $(i, j + 1)$ dont le premier nœud est justement propriétaire. Bien qu'aucun interblocage ne soit à craindre avec le schéma adopté dans les algorithmes de *change_mapping* et de *change_distribution*, il peut donc arriver que le partitionnement d'une matrice soit tel que, pendant certaines phases de la redistribution,

7. La bibliothèque POM et la classe POM ont été décrites au paragraphe 1.2.4.

la quasi-totalité des nœuds soient bloqués temporairement en attente chacun d'un bloc particulier.

On pourrait atténuer les contraintes de synchronisation imposées par l'alternance des émissions (non bloquantes) et des réceptions (bloquantes) en utilisant par exemple les primitives de la classe POM permettant de tester les files de réception. Les algorithmes de *change_mapping* et de *change_distribution* devraient alors être réécrits de telle sorte que chaque nœud procède à l'émission en séquence des blocs dont il est propriétaire (en testant entre deux émissions l'état des files de réception), et n'interrompe ses émissions que pour consommer les blocs disponibles dans ses files de réception. Avec cette approche, il y aurait toutefois un risque de saturer sur certaines machines parallèles les canaux de communication et/ou les files de réception des nœuds physique (la bibliothèque POM n'assure pas le contrôle de flux). Une approche plus raisonnable serait probablement de gérer au niveau de chaque nœud des *fenêtres d'émission*, chaque nœud étant alors en mesure d'expédier en séquence un certain nombre de ses blocs avant de se mettre en attente des blocs provenant des autres nœuds. La taille de la fenêtre d'émission pourrait éventuellement être ajustée en fonction des caractéristiques de la plate-forme utilisée.

5.1.1.4 Sélection dynamique des routines de redistribution

Il n'est pas très souhaitable d'imposer à un utilisateur de spécifier explicitement laquelle des routines *change_mapping* ou *change_distribution* doit être exécutée pour redistribuer une matrice de type DBLOCK_MATRIX. On peut toutefois s'inspirer des techniques décrites au paragraphe 4.4 pour assurer la sélection automatique de la routine appropriée.

La routine *redistribute* reproduite dans l'exemple 5.3 a pour rôle d'assurer dynamiquement la sélection transparente de la routine la plus appropriée pour redistribuer un objet de type DBLOCK_MATRIX. Elle fournit donc à l'utilisateur un moyen de redistribuer aisément toute instance de la classe DBLOCK_MATRIX, comme illustré dans le petit programme d'application SPMD reproduit ci-dessous.

Imaginons que dans le programme de l'exemple 5.4 les contraintes du calcul imposent que les deux matrices A et B soient distribuées différemment au cours de la première partie de l'exécution du programme, alors que dans la seconde partie elles doivent être distribuées de manière identique. Il suffit d'invoquer la routine *redistribute* de la classe DBLOCK_MATRIX pour redistribuer dynamiquement la matrice B et lui faire adopter le même schéma de distribution que celui de la matrice A.

Dans l'exemple 5.4, les matrices A et B n'ont pas le même partitionnement (la matrice A est partitionnée en blocs de taille 5×2 et la matrice B en blocs de taille 7×3). Lors de l'exécution de *redistribute*, la routine *change_distribution* va donc être automatiquement sélectionnée pour procéder à la redistribution requise.

Exemple 5.3

```
class DBLOCK_MATRIX inherit
    ...
feature -- Matrix redistribution
    redistribute (new_dist: DISTRIBUTION_2D) is
        do                                                          5
            if ((dist.bfi = new_dist.bfi)
                 and (dist.bfj = new_dist.bfj)) then
                change_mapping (new_dist)
            else
                change_distribution (new_dist);                     10
            end; -- if
        end; -- redistribute
    ...
end -- DBLOCK_MATRIX
```

Exemple 5.4

```
        local
            A, B: MATRIX;
        do
           !DBLOCK_MATRIX!A.make (50, 50, 5, 2, ROW_WISE_MAPPING);
           !DBLOCK_MATRIX!B.make (50, 50, 7, 3, COLUMN_WISE_MAPPING); 5
            ...(1)...
            B.redistribute (A.dist);
            ...(2)...
        end;
```

5.1.2 Cas des autres matrices distribuées

En développant Paladin, nous avons également doté les classes DCOL_MATRIX et DROW_MATRIX d'une version propre de la routine *redistribute*. Dans chacune de ces deux classes, l'algorithme de redistribution est implanté en tenant compte des caractéristiques de distribution propres à la classe considérée. Ainsi dans la classe DCOL_MATRIX les données échangées par les nœuds au cours de la redistribution sont des vecteurs colonnes, alors qu'il s'agit de vecteurs lignes dans la classe DROW_MATRIX. Il est à noter que lorsqu'une matrice distribuée par lignes ou par colonnes doit être redistribuée, seul le placement des vecteurs peut changer au cours de la redistribution. L'algorithme encapsulé dans la routine *redistribute* dans chacune des deux classes s'apparente donc beaucoup à celui de *change_mapping* dans la classe DBLOCK_MATRIX.

Nous n'avons pu définir la routine *redistribute* dans la classe DIST_MATRIX. En effet, la redistribution implique que l'on soit en mesure de désigner la table des blocs de la matrice courante, pour pouvoir éventuellement la remplacer par une nouvelle table à l'issue de la redistribution (voir par exemple la dernière ligne de l'exemple 5.2). Or, au niveau de la classe abstraite DIST_MATRIX, la notion de table des blocs est absente. Nous avons cependant déclaré — sans toutefois la définir — la routine *redistribute* dans la classe DIST_MATRIX. Cette déclaration impose que toute classe descendante de DIST_MATRIX fournisse une mise en œuvre pour la routine *redistribute*, mais elle garantit du même coup que toute matrice distribuée est redistribuable.

5.2 Changement du type d'une matrice

5.2.1 Contrainte due au typage statique

Comme la plupart des langages à objets à typage statique strict, le langage Eiffel ne permet pas que le type d'un objet soit modifié dynamiquement : dès lors qu'un objet a été créé avec un type donné, il ne peut plus en changer. Dans Paladin, il est donc par exemple impossible de transformer un objet de type LOCAL_MATRIX en un objet de type DBLOCK_MATRIX. Toutefois, on peut contourner cette contrainte et proposer une approximation satisfaisante de « polymorphisme » des matrices en utilisant les seules choses qui soient réellement polymorphes dans le langage Eiffel : les entités [8].

8. Les entités sont en Eiffel les attributs des classes, les variables locales et les arguments formels des routines, et l'entité prédéfinie **Result** dans les fonctions.

Point de langage 5.1

Règle de compatibilité des types. La règle de compatibilité des types autorise une affectation de la forme $a := b$ non seulement si a et b sont du même type, mais plus généralement si a et b sont de types A et B tels que B est un descendant de A. Cela correspond à l'idée intuitive qu'une valeur d'un type plus spécialisé peut être affecté à une entité d'un type moins spécialisé, mais pas le contraire.

En Eiffel, une entité est en général capable de prendre plusieurs types dynamiques, c'est-à-dire capable de référencer à l'exécution des objets de plus d'un type. Les règles de conformité du typage (point de langage 5.1) imposent que les types dynamiques possibles pour une entité x soient tous conformes au type statique de x (c'est-à-dire son type déclaré). C'est de cette manière que le polymorphisme est maintenu sous contrôle par le système de typage.

5.2.2 Changement de type réalisé dans un programme d'application

Lorsqu'on désire changer dans Paladin le type d'un agrégat matrice référencé par une entité x, on peut procéder à une conversion qui s'effectue en trois temps. Dans un premier temps, un nouvel objet est créé avec le type désiré. Ensuite, les données contenues dans l'agrégat source sont « transférées » dans le nouvel agrégat. Enfin, on fait en sorte que l'entité x — qui référençait jusqu'à présent l'objet agrégat source — référence à présent le nouvel objet agrégat. Cette procédure de conversion est illustrée dans l'exemple 5.5 et dans la figure 5.2.

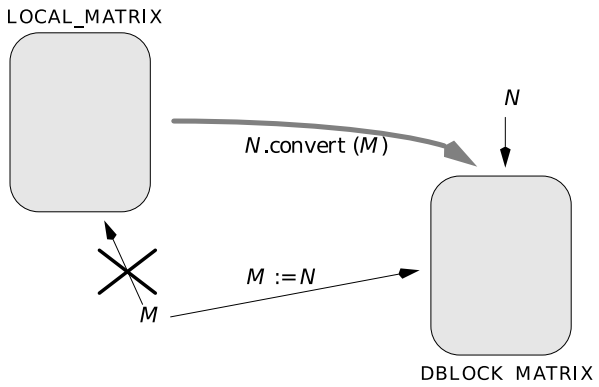


FIG. 5.2 - *Conversion de matrice avec changement de type*

Supposons que dans un programme d'application une matrice locale soit créée et affectée à la variable *M*. Après une première phase de calculs (notée `(1)` dans l'exemple 5.5), il devient nécessaire de transformer cette matrice locale en une matrice distribuée par blocs. La conversion du type de la matrice référencée par x s'effectue comme suit :

1. Une instance de la classe DBLOCK_MATRIX est créée et affectée à une variable temporaire *N*.

2. Les données contenues dans la matrice *M* sont transférées de la matrice locale *M* vers la nouvelle matrice *N*. La routine *convert* décrite plus loin permet de réaliser ce transfert de données.

3. Lorsque le transfert est terminé, on affecte à la variable *M* le nouvel agrégat (c'est-à-dire la matrice distribuée référencée par *N*) en procédant à une *affectation polymorphe*. La variable *M* conserve le même type statique MATRIX, mais son type dynamique (celui de l'objet référencé par la variable) a changé, passant de LOCAL_MATRIX a DBLOCK_MATRIX.

À l'issue de ces trois opérations, la matrice référencée par *M* — initialement une matrice locale — est « devenue » une matrice distribuée. L'exécution du programme d'application peut se poursuivre, la matrice référencée par *M* ayant changé de type mais étant toujours la « même » matrice (*i.e.* mêmes dimensions et même contenu) du point de vue abstrait de l'utilisateur.

Exemple 5.5

```
local
    M, N : MATRIX ;
do
   !LOCAL_MATRIX !M.make (...);
    ...(1)...                                          5
   !DBLOCK_MATRIX !N.make (...);
    N.convert (M);
    M := N ;
    ...(2)...
end ;                                                  10
```

La routine convert

Conceptuellement, la routine *convert* procède simplement à la recopie des données depuis une matrice source vers une matrice cible (qu'elles soient de même

type ou non). Elle requiert simplement que les deux matrices impliquées dans la conversion soient de même taille. Dans la classe MATRIX, nous avons doté la routine *convert* d'une mise en œuvre très simple, basée sur deux boucles imbriquées et des appels aux accesseurs de base *put* et *item*. Cependant, il ne s'agit là que d'une mise en œuvre par défaut. De meilleures versions de la routine *convert* ont été encapsulées dans certaines des classes descendant de MATRIX. Pour développer ces versions optimisées de la routine *convert*, nous avons notamment mis en pratique certaines des techniques d'optimisation évoquées au chapitre 4.

Discussion

La méthode proposée ici pour changer dynamiquement le type d'un agrégat nécessite qu'un nouvel objet soit créé avec les caractéristiques désirées (format de représentation interne, schéma de distribution, etc.). Cette approche demeure raisonnable dans la mesure où le ramasse-miettes assure qu'après la conversion de type l'agrégat source, devenu obsolète, pourra être récupéré par le ramasse-miettes.

L'un des problèmes majeurs soulevés par cette approche réside toutefois dans le manque de transparence de la conversion vis à vis de l'utilisateur. Celui-ci doit en effet déclarer explicitement dans un programme d'application une variable (ou un attribut) destiné à référencer temporairement un nouvel agrégat, créer cet objet avec le type désiré, invoquer la routine *convert* sur cet agrégat et finalement faire en sorte que l'entité qui référençait l'agrégat original référence à présent le nouvel agrégat. Le langage Eiffel n'offre apparemment aucune manière élégante d'encapsuler toutes ces opérations dans une seule expression. Il serait par contre envisageable de développer un petit outil (en fait une sorte de pré-compilateur semblable à l'outil *cpp* associé à la plupart des compilateurs C actuels), capable de réaliser les trois opérations élémentaires de la conversion de type avant la compilation Eiffel proprement dite. Un tel outil pourrait alors être intégré dans la boîte à outils d'EPEE.

Un autre problème majeur est celui des *alias*. Il peut arriver que dans un programme d'application on référence une même matrice plusieurs fois à travers plusieurs entités, ou bien encore qu'on accède à la fois à un agrégat matrice tout entier et à certaines sections de cette matrice (grâce au mécanisme des vues, par exemple). La technique de changement de type proposée plus haut ne tient absolument aucun compte des *alias* possibles sur la matrice soumise à la conversion de type.

Il existe des méthodes connues pour tenter de résoudre ce problème. Une première méthode consiste à gérer une table référençant tous les objets susceptibles de changer de type dynamiquement, et à s'imposer de n'accéder aux objets qu'à

travers cette table[9]. Une autre méthode consiste à maintenir dans chaque objet polymorphe une liste de références vers tous les objets qui référencent l'objet lui-même. Quand cet objet est soumis à une conversion de type, il doit en informer tous les objets référencés dans la liste en invoquant sur chacun de ces objets une routine particulière à laquelle il passe en paramètre sa nouvelle identité (en l'occurrence une référence sur lui-même).

Aucune de ces deux méthodes n'est franchement satisfaisante. La première méthode impose un surcoût en termes d'occupation de la mémoire (il faut créer et maintenir à jour la table d'indirections) et en termes de temps d'accès aux objets polymorphes (il faut franchir une indirection supplémentaire pour chaque accès). La seconde méthode impose également un surcoût en termes d'occupation de la mémoire (il faut maintenir une liste de références dans les objets polymorphes et des attributs supplémentaires dans les objets référençant des objets polymorphes). Elle est de surcroît particulièrement difficile à mettre en œuvre.

Dans l'état actuel de Paladin, un agrégat matrice (ou vecteur) peut donc changer de type dynamiquement mais la procédure de changement de type demeure assez fastidieuse pour l'utilisateur. Le maintien de la cohérence des alias n'est en outre pas assuré. Nous présentons dans le paragraphe suivant une approche alternative à la conversion de type qui pourrait être mise en œuvre dans Paladin au prix de quelques modifications dans la hiérarchie des classes.

5.2.3 Encapsulation du changement de type

On montre dans ce paragraphe comment on peut développer de véritables agrégats polymorphes à l'aide d'un langage à objets à typage statique tel que Eiffel, en illustrant notre propos avec l'exemple des agrégats matrices de Paladin.

5.2.3.1 Principe

L'idée fondamentale est d'introduire une distinction entre le concept d'objet matrice et celui d'objet « contenant les données d'une matrice ». Une matrice est alors un objet qui, au lieu de stocker directement ses données, fait référence à un objet *conteneur*. Une matrice polymorphe est donc une matrice capable de changer de conteneur dynamiquement et de manière transparente pour l'utilisateur.

5.2.3.2 Mise en œuvre

On pourrait développer une nouvelle classe MATRIX_CONTAINER ainsi que des classes descendantes, et introduire dans la classe MATRIX actuelle un attribut per-

9. En Smalltalk, tous les objets sont gérés ainsi, chaque entrée dans la table d'indirection jouant le rôle de *handle* pour un objet particulier.

mettant à toute matrice de référencer un objet de type conforme au type MATRIX_CONTAINER. Cependant cette approche obligerait à modifier la mise en œuvre de la plupart des routines définies dans la classe MATRIX, et il faudrait en outre apporter d'importantes modifications à l'ensemble des classes décrivant les matrices dans la hiérarchie actuelle de Paladin.

Il est beaucoup plus judicieux procéder de manière inverse, et de considérer la classe MATRIX actuelle et ses descendantes comme constituant la hiérarchie décrivant les objets conteneurs. Il suffit alors de construire une seule et unique nouvelle classe — appelons la POLY_MATRIX — caractérisant les matrices polymorphes.

La classe POLY_MATRIX adopte une interface semblable à celle de la classe MATRIX, mais on la dote d'un attribut supplémentaire *container* permettant à tout objet de type POLY_MATRIX de référencer un objet de type MATRIX (voir l'exemple 5.6).

Exemple 5.6

```
class POLY_MATRIX

feature {NONE} -- Reference to a matrix container
    container : MATRIX ;

                                                                    5
feature -- Basic Accessors
    item (i, j: INTEGER): DOUBLE is do Result := container.item (i, j); end ;
    put (v: like item ; i, j: INTEGER) is do container.put (v, i, j); end ;
    ...
feature -- Operators                                                10
    mult (A, B: like Current) is
        do
            container.mult (A.container, B.container);
        end ;
    ...                                                             15
end -- POLY_MATRIX
```

L'attribut *container* étant encapsulé dans la classe POLY_MATRIX, il n'apparaît pas dans son interface et le changement de conteneur peut donc être maintenu transparent pour l'utilisateur.

Les accesseurs *put* et *item* de la classe POLY_MATRIX sont définis de manière à accéder aux données stockées dans le conteneur associé à la matrice polymorphe courante. De même, on implante chaque opérateur déclaré dans la classe POLY_MATRIX de manière à ce qu'il invoque simplement l'opérateur homonyme de la classe

MATRIX. Par exemple, l'opérateur *mult* de la classe POLY_MATRIX a pour seul rôle d'invoquer l'opérateur *mult* de la classe MATRIX avec les paramètres adéquats (voir l'exemple 5.6). En définissant ainsi chacun des opérateurs de la classe POLY_MATRIX, la dégradation des performances due à l'indirection imposée par l'attribut *container* est maintenue à un niveau insignifiant.

La classe POLY_MATRIX s'intègre dans la hiérarchie des classes de Paladin en devenant une classe *cliente* des classes pré-existantes (voir figure 5.3). Il n'y a donc pas lieu de modifier quoi que ce soit à la classe MATRIX ou à ses descendantes.

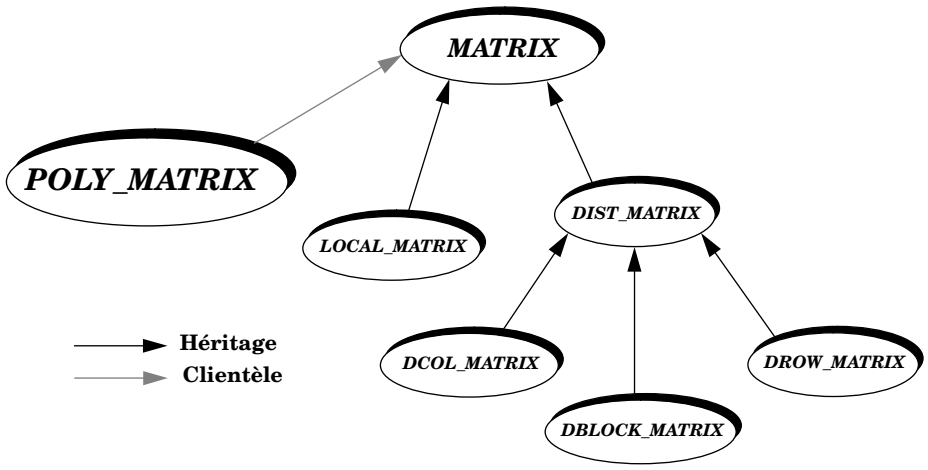


FIG. 5.3 - *Introduction de la classe* POLY_MATRIX *dans la hiérarchie des classes de Paladin*

Pour que les matrices polymorphes caractérisées par la classe POLY_MATRIX puissent changer de représentation interne dynamiquement — c'est-à-dire, pour que les objets de type POLY_MATRIX puissent substituer un nouveau conteneur au conteneur courant —, il suffit de doter la classe POLY_MATRIX de routines procédant aux substitutions désirées. Par exemple, pour qu'une matrice quelconque puisse être transformée dynamiquement en une matrice locale, on intègre dans la classe POLY_MATRIX la routine *become_local* reproduite dans l'exemple 5.7. Avec cette routine, la conversion d'une matrice polymorphe référencée par une entité M peut être obtenue avec la simple expression *M.become_local*.

Il faut de même définir dans la classe MATRIX plusieurs routines de création permettant de créer, en même temps qu'une instance de cette classe, un conteneur associé ayant le type et les caractéristiques choisies.

Exemple 5.7

```
class POLY_MATRIX
    ...
feature -- Internal representation conversion
    become_local is
        local
            new_container: like container;
        do
            -- Create a new matrix container with type as required
            !LOCAL_MATRIX !new_container.make (nrow, ncolumn);
            -- Transfer data from old matrix container to new one
            new_container.convert (container);
            -- Adopt new matrix container and discard old one
            container := new_container;
        end; -- become_local
    ...
end -- class POLY_MATRIX
```

Discussion

L'approche décrite dans ce paragraphe présente quelques inconvénients : elle oblige à modifier la procédure de création des matrices de manière fondamentale. En effet, on doit définir dans la classe POLY_MATRIX un grand nombre de routines de création et de changement de format. En outre, à chaque apparition d'une nouvelle classe décrivant un format de représentation original pour les conteneurs, on doit mettre à jour la classe POLY_MATRIX en conséquence. Si on développait par exemple une classe SPARSE_MATRIX proposant un format de représentation interne pour les matrices creuses, il faudrait en même temps doter la classe POLY_MATRIX de deux nouvelles routines (*e.g. make_sparse* et *become_sparse*).

L'extension algorithmique de la bibliothèque Paladin est également rendue plus complexe : tout nouvel opérateur servant à manipuler des matrices doit être défini au moins deux fois : une première fois dans la classe MATRIX (où on le dote d'une mise en œuvre séquentielle par défaut), et une seconde fois dans la classe POLY_MATRIX (où on l'implante de manière à invoquer automatiquement l'opérateur homonyme de la classe MATRIX).

Ces inconvénients sont cependant largement compensés par les atouts de la classe POLY_MATRIX. Les objets de type POLY_MATRIX sont capables de changer

de format dynamiquement, sans risque d'incohérence des alias[10] (on a vu que la méthode présentée au paragraphe 5.2.2 pêche justement sur ce point). Ces objets sont en outre capables de changer de forme *spontanément*, c'est-à-dire *sans que l'utilisateur en fasse la demande* : on pourrait par exemple faire en sorte que la routine *become_local* soit invoquée dans l'une des routines de la classe POLY_MATRIX, ce qui aurait pour conséquence le changement « spontané » (du point de vue de l'utilisateur) du format de la matrice courante. La méthode présentée au paragraphe 5.2.2 ne permet pas d'obtenir de tels changements de type spontanés de la part des objets polymorphes.

Cette aptitude des objets de type POLY_MATRIX à changer de forme sans intervention de l'utilisateur ouvre des perspectives particulièrement intéressantes, car elle nous permet d'envisager de libérer l'utilisateur de la responsabilité du choix du format (et, le cas échéant, de la distribution) de chaque matrice. Il devient en effet possible de bâtir une bibliothèque dans laquelle les matrices changent de format spontanément en fonction des besoins du calcul en cours. Cette perspective est évoquée plus en détails dans le chapitre 7.

Travail connexe

Le problème de la mise en œuvre d'objets capables de changer de type dynamiquement a été également abordé par T. R. Davis pour les objets du langage C++ [40]. Dans cet article, l'auteur propose d'identifier dans chaque type d'objet capable de changer d'état les informations devant être affectées par un tel changement, et d'extraire ces informations de l'objet lui-même afin de les encapsuler dans un *méta-objet*, caractérisé par une *méta-classe* (les méta-classes décrites par T. R. Davis n'ont cependant rien à voir avec celles de Smalltalk).

10. À condition bien sûr que les références multiples à un même objet portent sur un objet de type POLY_MATRIX, et non sur un objet conteneur de type MATRIX.

Chapitre 6

Expérimentation

6.1 Généralités sur les mesures effectuées

Les mesures de performances dont les résultats sont rapportés dans les paragraphes qui suivent ont été réalisées en exécutant certains des opérateurs SPMD de la bibliothèque Paladin.

Pour chaque série de mesures, on présente les *accélérations* (ou *speedup*) observées, et l'*efficacité* de la parallélisation. On définit l'accélération Acc comme étant le résultat du rapport

$$Acc = \frac{Ts}{Tp}$$

où Ts désigne le temps d'exécution d'un algorithme séquentiel et Tp est le temps d'exécution d'un algorithme parallèle équivalent.

Afin de pouvoir comparer les performances indépendamment du nombre de processeurs utilisés, on indique également l'efficacité Eff, définie comme suit :

$$Eff = \frac{Acc}{P}$$

où P est le nombre de processeurs participant au calcul parallèle.

Le paragraphe 6.2 présente les résultats obtenus sur le réseau de stations de travail du projet Pampa, et le paragraphe 6.3 les résultats obtenus sur la machine Paragon XP/S de l'IRISA. Sur ces deux types de plate-forme, des mesures ont été réalisées *avec* et *sans* utilisation des routines du noyau BLAS.

On rappelle par ailleurs que, dans sa version actuelle, la bibliothèque Paladin ne manipule que des vecteurs et matrices de nombres réels représentés en *double*

précision. Tous les résultats rapportés dans les paragraphes suivants concernent donc des calculs réalisés en double précision.

6.2 Expérimentation sur réseau de stations de travail

6.2.1 Conditions d'expérimentation

Les résultats rapportés dans ce paragraphe ont été obtenus en réalisant des mesures sur un réseau de stations de travail de type Sun 4/75 SPARCstation 2, dotées chacune d'au moins 32 Moctets de RAM (on s'est efforcé de ne procéder qu'à des mesures ne nécessitant pas d'accès à la mémoire de *swap*).

La bibliothèque de communication utilisée lors de ces mesures était la POM, dans sa version mise en œuvre au dessus de PVM (version 3.3.7). Les compilateurs utilisés pour générer le code exécutable étaient, d'une part le compilateur Eiffel d'ISE version 3.2.3, d'autre part le compilateur *gcc* version 2.6.2 avec l'option d'optimisation `-O2`. Au cours des mesures, le contrôle assertionnel était désactivé sur l'ensemble des classes Eiffel.

Bien que les mesures aient été réalisées pendant la nuit, les stations de travail utilisées n'étaient pas dissociées du réseau de communication global du laboratoire. En conséquence, ni le médium de communication (bus Ethernet de bande passante 10 Mbits/s), ni les processeurs, disques et ressources mémoire des stations n'étaient entièrement disponibles pour les mesures. Les processus fonctionnant en tâches de fond sur les stations, et l'activité qu'ils induisent sur le réseau Ethernet, ont donc probablement influencé légèrement les résultats des mesures.

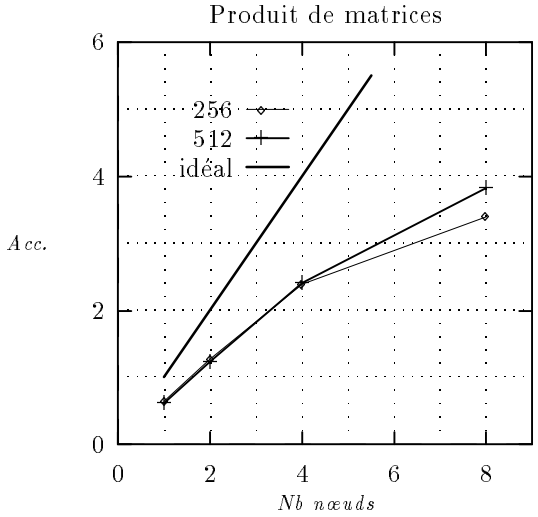
6.2.2 Expérimentation sans utilisation du noyau BLAS

Dans ce paragraphe sont rapportés les résultats des mesures effectuées avec la bibliothèque Paladin, sans qu'aucune routine du noyau BLAS soit invoquée au cours des calculs. Les algorithmes parallèles dont les performances sont rapportées ici sont donc des algorithmes 100 % Eiffel.

6.2.2.1 Produit de matrices distribuées par blocs

La figure 6.1 présente les accélérations et efficacités observées en calculant des produits de matrices carrées $C = A \times B$, les matrices A, B et C étant distribuées par blocs (objets de type DBLOCK_MATRIX). Pour réaliser ces mesures, on a utilisé l'opérateur *mult_dblock_dblock* reproduit dans l'exemple 4.7 (page 122).

Pour évaluer les accélérations, on a pris comme temps de référence les durées d'exécution observées avec l'opérateur séquentiel *mult*[1] de la classe MATRIX, opérant sur des matrices locales (instances de la classe LOCAL_MATRIX). Dans la figure 6.1 on peut voir les résultats obtenus pour des produits de matrices carrées de taille 256×256 et 512×512. On constate que, lorsqu'on fait varier le nombre de stations participant au calcul de 1 à 8, l'efficacité diminue rapidement pour atteindre environ 45 % sur 8 nœuds.



P	*temps (sec)*	*accélération*	*efficacité*
	$N = 256$		
Ref.	120.32	—	—
1	183.93	.65	65 %
2	93.55	1.28	64 %
4	50.17	2.39	59 %
8	35.33	3.40	42 %
	$N = 512$		
Ref.	925.83	—	—
1	1508.11	.61	61 %
2	752.15	1.23	61 %
4	383.13	2.41	60 %
8	241.63	3.83	47 %

FIG. 6.1 - *Produits de matrices distribuées par blocs*

Pour voir comment les performances du code Eiffel se situent par rapport à celles d'un code C équivalent, nous avons développé un programme C séquentiel calculant le produit de matrices : ce programme C réalise le produit de matrices 256×256 en 73.97 secondes (soit 1.63 fois plus vite que notre opérateur séquentiel *mult*), et le produit de matrices 512×512 en 640.82 secondes (soit 1.44 fois plus vite que l'opérateur *mult*).

Nous avons également procédé à quelques expériences en compilant un sous-ensemble des classes de Paladin avec le nouveau compilateur TowerEiffel version 1.4.3. Avec ce compilateur, la durée d'exécution de notre opérateur Eiffel séquentiel *mult* est en moyenne 12 % supérieure à celle du programme C séquentiel équivalent.

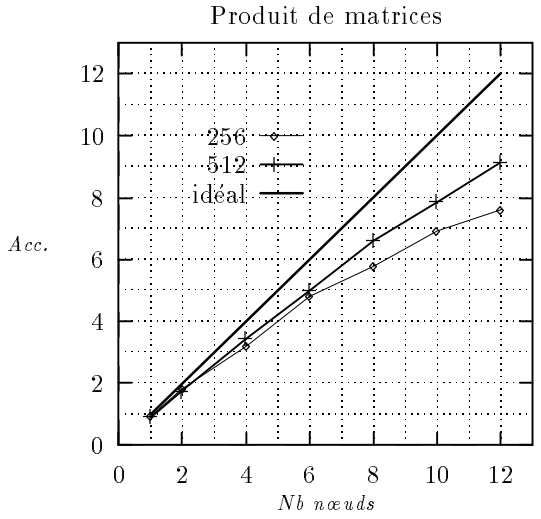
1. L'opérateur *mult* a été décrit au § 2.2.3.

6.2.2.2 Produit de matrices distribuées par lignes et par colonnes

La figure 6.2 présente les accélérations et efficacités observées en réalisant le même genre de produit de matrices carrées $C = A \times B$ qu'au paragraphe 6.2.2.1, la matrice A étant cette fois distribuée par lignes entrelacées (objet de type DROW_MATRIX avec le facteur de partitionnement *bfi = 1*), et les matrices B et C étant distribuées par colonnes entrelacées (objets de type DCOL_MATRIX avec le facteur de partitionnement *bfj = 1*).

Pour réaliser ces mesures, on a utilisé l'opérateur *mult_drow_dcol* défini dans la classe DCOL_MATRIX de Paladin. Cet opérateur n'a pas été décrit dans ce document, mais il ressemble beaucoup à l'opérateur *mult_dblock_dblock* décrit au paragraphe 4.2.3, excepté que les opérations élémentaires réalisées au cœur du nid de boucles ne sont pas des produits de blocs matrices, mais des produits scalaires de vecteurs locaux.

Pour évaluer les accélérations, on a pris les mêmes temps de référence que dans le paragraphe 6.2.2.1.



P	*temps (sec)*	*accélération*	*efficacité*
	$N = 256$		
Ref.	120.32	—	—
1	128.09	.93	93 %
2	66.22	1.81	90 %
4	37.56	3.20	80 %
6	24.89	4.83	80 %
8	20.79	5.78	72 %
10	17.41	6.91	69 %
12	15.83	7.60	63 %
	$N = 512$		
Ref.	925.83	—	—
1	1015.40	.91	91 %
2	531.07	1.74	87 %
4	267.53	3.46	86 %
6	184.86	5.00	83 %
8	139.98	6.61	82 %
10	117.51	7.87	78 %
12	101.50	9.12	76 %

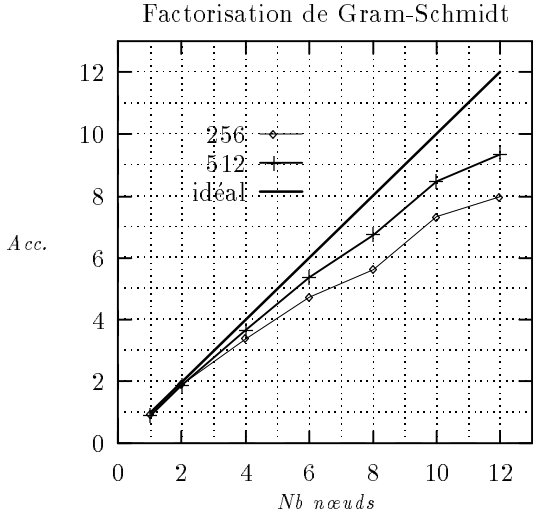
FIG. 6.2 - *Produits de matrices distribuées par lignes et par colonnes*

Dans la figure 6.2 on peut voir les résultats obtenus pour des produits de matrices de taille 256×256 et 512×512. On constate que, lorsqu'on fait varier le nombre de stations participant au calcul de 1 à 12, l'accélération demeure relativement linéaire et l'efficacité se maintient au dessus de 70 % (excepté dans le cas du produit de matrices 256×256 sur 10 et 12 stations). Ces résultats sont satisfaisants compte tenu du support d'expérimentation considéré ici.

6.2.2.3 Factorisation de Gram-Schmidt d'une matrice distribuée par colonnes

La figure 6.3 présente les accélérations et efficacités observées en réalisant des factorisations de type $A \rightarrow Q.R$ (avec Q recouvrant A) selon l'algorithme dit « de Gram-Schmidt modifié », en utilisant des matrices A et R distribuées par colonnes entrelacées (objets de type DCOL_MATRIX avec le facteur de partitionnement *bfj = 1*). Pour réaliser ces mesures, on a utilisé l'opérateur *mgs_dcol* décrit au paragraphe 4.2.3.

Pour évaluer les accélérations, on a pris comme temps de référence les durées d'exécution observées avec l'observateur séquentiel *mgs*[2] de la classe MATRIX.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
Ref.	149.25	—	—
1	155.47	.95	95 %
2	77.85	1.91	95 %
4	43.79	3.40	85 %
6	31.65	4.71	78 %
8	26.42	5.64	70 %
10	20.34	7.33	73 %
12	18.71	7.97	66 %
$N = 512$			
Ref.	1178.45	—	—
1	1295.54	.90	90 %
2	632.98	1.86	93 %
4	321.34	3.66	91 %
6	219.02	5.38	89 %
8	174.56	6.75	84 %
10	139.28	8.46	84 %
12	125.90	9.36	78 %

FIG. 6.3 - *Factorisations de Gram-Schmidt de matrices distribuées par colonnes*

On peut voir dans la figure 6.3 les résultats obtenus pour des matrices de taille 256×256 et 512×512. On constate que l'efficacité de la parallélisation se maintient au dessus de 70 %.

6.2.3 Expérimentation avec utilisation du noyau BLAS

Certaines des expériences rapportées précédemment ont été répétées en interfaçant la bibliothèque Paladin avec le noyau BLAS encapsulé dans la bibliothèque NAG[3].

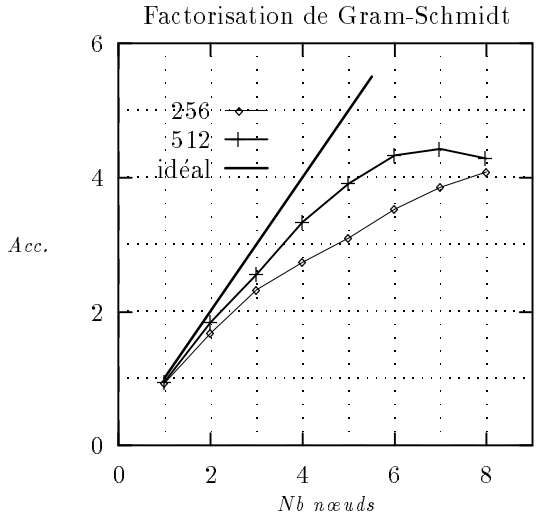
2. L'opérateur *mgs* a été décrit au § 2.2.3.

3. Bibliothèque `libnag.a` disponible dans l'environnement logiciel de l'IRISA au 08/04/95.

6.2.3.1 Factorisation de Gram-Schmidt d'une matrice distribuée par colonnes

On a répété l'expérience de factorisation de Gram-Schmidt décrite au paragraphe 6.2.2.3, les opérations vecteur-vecteur réalisées au cours de l'exécution de l'algorithme se traduisant cette fois par l'appel de routines de type BLAS-1.

Pour évaluer les accélérations, on a pris comme temps de référence les durées d'exécution observées avec un opérateur Eiffel séquentiel faisant appel aux mêmes routines de niveau BLAS-1 que l'algorithme de l'opérateur SPMD *mgs_dcol*.



P	*temps (sec)*	*accélération*	*efficacité*
	$N = 256$		
Ref.	27.42	—	—
1	29.40	.93	93 %
2	16.39	1.67	83 %
3	11.77	2.32	77 %
4	9.99	2.74	68 %
5	8.82	3.10	62 %
6	7.75	3.53	58 %
7	7.12	3.85	55 %
8	6.72	4.08	51 %
	$N = 512$		
Ref.	222.37	—	—
1	231.67	.95	95 %
2	121.22	1.83	91 %
3	87.15	2.55	85 %
4	66.70	3.33	83 %
5	56.87	3.91	78 %
6	51.33	4.33	72 %
7	50.26	4.42	63 %
8	51.94	4.28	53 %

FIG. 6.4 - *Factorisations de Gram-Schmidt ds matrices distribuées par colonnes, avec appels à des routines de niveau BLAS-1*

On voit dans la figure 6.4 qu'avec l'utilisation des routines du noyau BLAS, l'accélération due à l'exécution en parallèle n'est intéressante que pour un nombre de stations faible. L'efficacité, initialement supérieure à 90 %, retombe très vite pour approcher 50 % dans le cas d'une exécution sur 8 stations de travail.

Ce phénomène est caractéristique d'un rapport calcul/communication défavorable. Lorsqu'on exécute sur réseau de stations de travail des opérateurs SPMD procédant à des diffusions de vecteurs et invoquant des routines de niveau BLAS-1 pour réaliser les calculs locaux, le coût des communications sur le médium Ethernet devient vite trop prohibitif par rapport au coût des calculs réalisés sur chaque station.

6.3 Expérimentation sur la machine Intel Paragon XP/S

6.3.1 Conditions d'expérimentation

Les résultats rapportés dans ce paragraphe ont été obtenus en réalisant des mesures sur la machine Intel Paragon XP/S de l'IRISA, fonctionnant avec le système d'exploitation Mach OSF/1 (version 1.0.4). La bibliothèque de communication utilisée était la POM, dans sa version mise en œuvre au dessus des primitives du noyau NX/2.

Les compilateurs utilisés pour générer le code exécutable étaient, d'une part le compilateur Eiffel d'ISE version 3.2, d'autre part le compilateur *icc*[4] version R4.5 avec l'option d'optimisation *-O2*.

6.3.2 Expérimentation sans utilisation du noyau BLAS

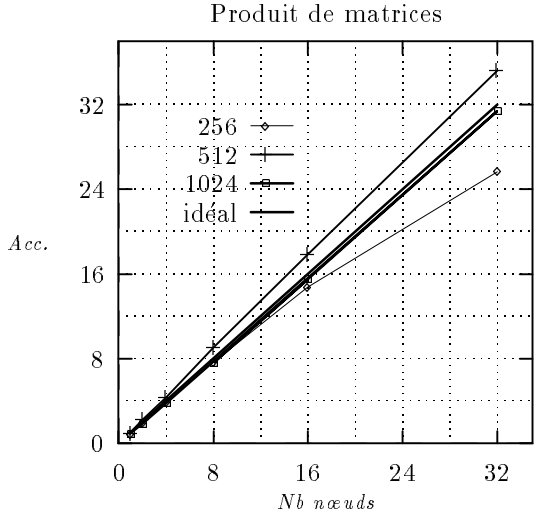
6.3.2.1 Produits de matrices distribuées par blocs

La figure 6.5 présente les accélérations et efficacités observées en calculant des produits de matrices carrées $C = A \times B$, les matrices A, B et C étant distribuées par blocs (objets de type DBLOCK_MATRIX). Pour réaliser ces mesures, on a utilisé l'opérateur *mult_dblock_dblock* reproduit dans l'exemple 4.7 (page 122).

Pour évaluer les accélérations, on a pris comme temps de référence les durées d'exécution observées avec l'opérateur séquentiel *mult*[5] de la classe MATRIX, opérant sur des matrices locales (instances de la classe LOCAL_MATRIX). Dans la figure 6.5 on peut voir les résultats obtenus pour des produits de matrices carrées de taille 256×256, 512×512 et 1024×1024. Pour le produit de matrices 1024×1024, on n'a pu réaliser des mesures qu'à partir de 8 nœuds (en raison de l'espace mémoire important requis pour stocker en mémoire trois matrices de taille 1024×1024). Pour cette série de mesures le temps de référence a donc été obtenu par extrapolation.

On constate que l'efficacité demeure supérieure à 90 %, et dépasse même 100 % dans le cas du produit de matrices 512×512. Il s'agit là d'une conséquence de la distribution des données sur une machine dans laquelle les accès à la mémoire sont particulièrement prohibitifs et parfois même plus coûteux que les échanges de données entre les nœuds. Dans ce cas précis, la distribution des matrices fait que chaque nœud a moins de données à gérer localement. L'économie ainsi réalisée sur le coût des accès à la mémoire locale (moins de défauts de cache) n'est pas

4. Compilateur développé pour la Paragon par *Intel Corporation* et *The Portland Group*.
5. L'opérateur *mult* a été décrit au § 2.2.3.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
Ref.	114.82	—	—
1	122.63	.93	93 %
2	58.77	1.95	97 %
4	29.12	3.94	98 %
8	14.90	7.70	96 %
16	7.78	14.75	92 %
32	4.46	25.74	80 %
$N = 512$			
Ref.	1042.67	—	—
1	1049.68	.99	99 %
2	460.36	2.26	113 %
4	238.14	4.37	109 %
8	115.21	9.05	113 %
16	58.27	17.89	111 %
32	29.59	35.23	110 %
$N = 1024$			
Ref.	7300	—	—
1	—	—	—
2	—	—	—
4	—	—	—
8	953.38	7.65	95 %
16	468.87	15.56	97 %
32	232.29	31.42	98 %

FIG. 6.5 - *Produits de matrices distribuées par blocs*

compensée par le coût des communications : l'accélération est supra-linéaire, c'est-à-dire meilleure que l'accélération « idéale ».

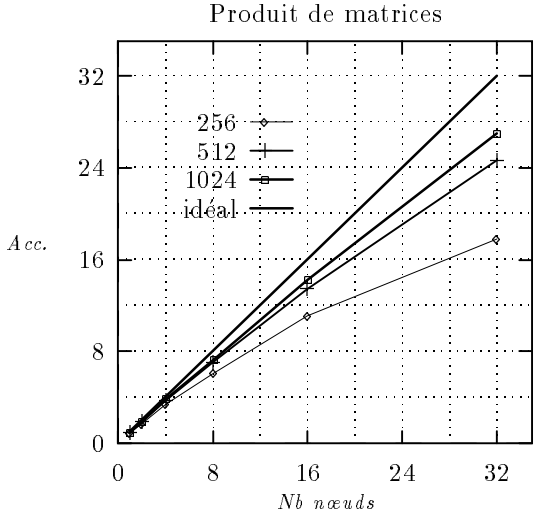
6.3.2.2 Produit de matrices distribuées par lignes et par colonnes

La figure 6.6 présente les accélérations et efficacités observées en réalisant le même genre de produit de matrices carrées $C = A \times B$ qu'au paragraphe 6.3.2.1, la matrice A étant cette fois distribuée par lignes entrelacées (objet de type DROW_MATRIX avec le facteur de partitionnement *bfi = 1*), et les matrices B et C étant distribuées par colonnes entrelacées (objets de type DCOL_MATRIX avec le facteur de partitionnement *bfj = 1*).

Pour réaliser ces mesures, on a utilisé l'opérateur *mult_drow_dcol* défini dans la classe DCOL_MATRIX de Paladin[6].

Pour évaluer les accélérations, on a pris les mêmes temps de référence que dans le paragraphe 6.3.2.1. Dans la figure 6.6, on peut voir les résultats obtenus pour des produits de matrices de taille 256×256, 512×512 et 1024×1024 (dans ce dernier

6. L'opérateur *mult_drow_dcol* a été décrit succinctement dans le § 6.2.2.2.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
Ref.	68.75	—	—
1	78.35	.87	87 %
2	39.65	1.73	86 %
4	20.33	3.38	84 %
8	11.23	6.12	76 %
16	6.21	11.07	69 %
32	3.87	17.76	55 %
$N = 512$			
Ref.	580.23	—	—
1	611.52	.94	94 %
2	307.34	1.88	94 %
4	155.38	3.73	93 %
8	82.17	7.06	88 %
16	43.06	13.47	84 %
32	23.55	24.63	76 %
$N = 1024$			
Ref.	4800	—	—
1	—	—	—
2	—	—	—
4	—	—	—
8	661.47	7.25	90 %
16	338.12	14.19	88 %
32	177.99	26.96	84 %

FIG. 6.6 - *Produits de matrices distribuées par lignes et par colonnes*

cas, on n'a pu réaliser des mesures qu'à partir de 8 nœuds). On peut constater que pour les matrices de grante taille, l'efficacité demeure supérieure à 70 %.

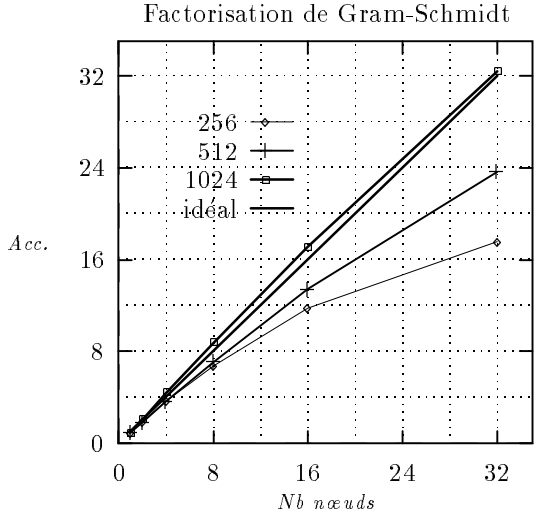
6.3.2.3 Factorisation de Gram-Schmidt d'une matrice distribuée par colonnes

La figure 6.7 présente les accélérations et efficacités observées en réalisant une factorisation de type $A \rightarrow Q.R$ (avec Q recouvrant A) selon l'algorithme dit « de Gram-Schmidt modifié », en utilisant des matrices A et R distribuées par colonnes entrelacées (objets de type DCOL_MATRIX avec le facteur de partitionnement *bfj = 1*). Pour réaliser ces mesures, on a utilisé l'opérateur *mgs_dcol* décrit au paragraphe 4.2.3.

Pour évaluer les accélérations, on a pris comme temps de référence les durées d'exécution observées avec l'observateur séquentiel *mgs*[7] de la classe MATRIX.

On voit dans la figure 6.7 les résultats obtenus pour des matrices de taille 256 ×

7. L'opérateur *mgs* a été décrit au § 2.2.3.



P	*temps (sec)*	*accélération*	*efficacité*
	$N = 256$		
Ref.	84.17	—	—
1	87.53	.96	96 %
2	44.42	1.89	94 %
4	22.87	3.68	92 %
8	12.45	6.76	84 %
16	7.16	11.75	73 %
32	4.79	17.57	54 %
	$N = 512$		
Ref.	654.21	—	—
1	698.09	.93	93 %
2	357.12	1.83	91 %
4	178.27	3.66	91 %
8	91.95	7.11	88 %
16	48.78	13.41	83 %
32	27.61	23.69	74 %
	$N = 1024$		
Ref.	6358.98	—	—
1	6692.34	.95	95 %
2	2998.96	2.12	106 %
4	1426.17	4.45	111 %
8	722.79	8.79	109 %
16	371.80	17.10	106 %
32	196.50	32.36	101 %

FIG. 6.7 - *Factorisations de Gram-Schmidt de matrices distribuée par colonnes*

256, 512×512 et 1024×1024. On constate que l'efficacité est très bonne pour les matrices de grande taille, et qu'elle dépasse encore très souvent 100 %.

6.3.3 Expérimentation avec utilisation du noyau BLAS

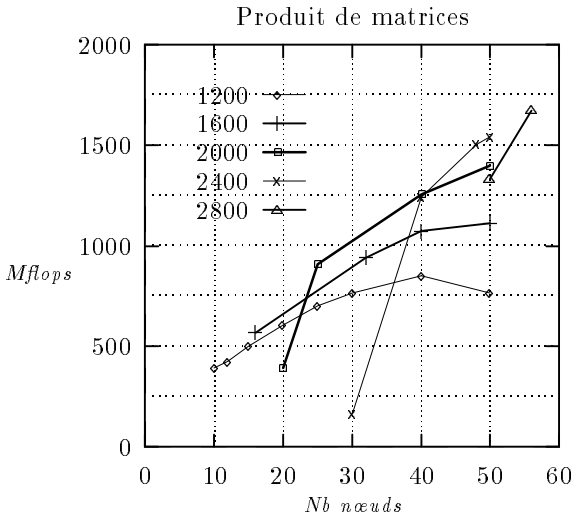
Certaines des expériences rapportées précédemment ont été répétées en interfaçant la bibliothèque Paladin avec le noyau BLAS encapsulé dans la bibliothèque mathématique[8] fournie avec la machine Paragon XP/S.

6.3.3.1 Produit de matrices distribuées par blocs

On rapporte dans ce paragraphe les performances observées en calculant des produits de matrices distribuées par blocs à l'aide de l'opérateur *mult_dblock_dblock* décrit au paragaphe 4.2.3, les opérations élémentaires de produits de blocs matrices étant réalisées en appelant la routine DGEMM[9] du noyau BLAS (routine de niveau BLAS-3).

8. Bibliothèque `libkmath.a` disponible dans l'environnement logiciel de la Paragon au 08/04/95.
9. DGEMM : *Double precision GEneral Matrix Multiply.*

Sur la machine Paragon XP/S, l'emploi de routines de niveau BLAS-3 augmente les performances des algorithmes d'algèbre linéaire de manière très sensible. En réalisant un produit de matrices avec un code généré à partir d'un programme source C ou Fortran, les performances observées sur un nœud de la machine ne dépassent pas 2 Mflops dans le meilleur des cas. En appelant pour effectuer le même calcul la routine DGEMM du noyau BLAS, les performances observées sur un nœud peuvent atteindre 45 Mflops (les performances effectives varient selon la taille des matrices impliquées dans le calcul).



P	*temps (sec)*	*Mflops*	*Mflops/nœud*
	$N = 1200$		
10	8.83	391.39	39.13
20	5.69	607.38	30.36
30	4.53	762.91	25.43
40	4.07	849.14	21.22
50	4.53	762.91	15.25
	$N = 1600$		
16	14.41	568.49	35.53
32	8.69	942.69	29.45
40	7.64	1072.25	26.80
50	7.36	1113.04	22.26
	$N = 2000$		
20	40.74	392.73	19.63
25	17.63	907.54	36.30
40	12.72	1257.86	31.44
50	11.45	1397.37	27.94
	$N = 2400$		
30	169.00	163.59	5.45
40	22.27	1241.49	31.03
48	18.39	1503.42	31.32
50	17.92	1542.85	30.85
	$N = 2800$		
50	32.96	1332.03	26.64
56	26.27	1671.25	29.84

FIG. 6.8 - *Produits de matrices distribuées par blocs, avec appels à la routine* DGEMM *de BLAS-3*

Lorsqu'on fait intervenir la routine DGEMM pour calculer les produits de matrices dans Paladin, le coût des communications cesse d'être négligeable par rapport à celui des calculs (alors que c'était le cas dans les expériences relatées au paragraphe 6.3.2.1). Il faut donc réaliser des produits de matrices de très grande taille pour que les performances observées soient intéressantes. Pour les matrices de petite taille, on ne gagne pas grand chose à distribuer les matrices puisque au delà d'une dizaine de nœuds participant au calcul, la parallélisation n'est plus efficace (la courbe d'accélération atteint un plafond, puis redescend très vite).

On a reproduit dans la figure 6.8 les *performances* observées (attention, il ne s'agit plus d'accélération !) pour diverses tailles de matrices carrées. Ces performances sont équivalentes à celles pouvant être obtenues avec d'autres approches, notamment avec un programme C ou Fortran parallélisé manuellement et faisant aussi intervenir la routine DGEMM de BLAS. On notera en particulier la performance d'environ 1.7 Gflops observée pour un produit de matrices 2800×2800 sur 56 nœuds.

Chapitre 7

Bilan et perspectives

7.1 Bilan

Le travail de thèse rapporté dans ce document s'est inscrit dans le cadre du développement et de l'expérimentation d'EPEE, un environnement pour la programmation par objets des architectures parallèles à mémoire distribuée (APMD). L'objectif du projet EPEE est de montrer qu'il est possible de développer à l'aide d'un langage à objets purement séquentiel des composants logiciels pour la programmation des architectures parallèles.

Dans les contributions de cette thèse, on peut distinguer :

- L'aspect méthodologique : j'ai introduit la notion d'agrégat polymorphe (des structures de données homogènes de grande taille capables de changer de forme, et notamment d'être distribuées sur une machine parallèle) et proposé une approche pour mettre en œuvre à l'aide des mécanismes de la programmation par objets des bibliothèques d'agrégats portables, extensibles, performantes et d'un emploi aisé ;

- L'illustration : j'ai conçu et développé une bibliothèque de démonstration, la bibliothèque Paladin, dédiée au calcul d'algèbre linéaire sur APMD. Cette bibliothèque est extensible, tant du point de vue algorithmique (ajout de nouveaux algorithmes séquentiels ou parallèles) que du point vue du polymorphisme des agrégats matrices et vecteurs (ajout de nouveaux formats de représentation en mémoire et/ou de distribution). La distribution des données et l'exécution parallèle des calculs demeurent transparents pour l'utilisateur. Les performances observées sont satisfaisantes, et la portabilité de la bibliothèque est assurée. Ces résultats confirment que l'approche consistant à utiliser les mécanismes de la programmation par objets pour construire des bibliothèques pour machines parallèles est une approche viable.

La bibliothèque Paladin a fait l'objet de plusieurs publications [60, 81], et de présentations dans des conférences internationales [66, 61, 58] et nationale [59].

- L'extension d'EPEE : au cours du développement de la bibliothèque Paladin, j'ai été amené à développer un certain nombre de mécanismes génériques pour aider à la distribution des données et à la parallélisation des calculs. Ces mécanismes ont été intégrés à la boîte à outils de l'environnement EPEE. J'ai également contribué à la conception et à la mise en œuvre au sein du projet Pampa d'une bibliothèque de communication dotée de mécanismes d'observation : la bibliothèque POM [11, 67, 68, 69]. Cette bibliothèque a également été intégrée à l'environnement EPEE. Elle assure à présent la complète portabilité des applications parallèles développées avec cet environnement.

Le portage de l'environnement EPEE et l'expérimentation de la bibliothèque Paladin sur le super-calculateur PARAGON XP/S de l'IRISA ont été effectués dans le cadre d'un projet de collaboration en ERDP[1] entre l'INRIA et la société Intel SSD. Ce travail a fait l'objet de deux rapports de contrat [64, 65].

7.2 Perspectives

Les résultats encourageants obtenus avec la bibliothèque de démonstration Paladin nous incitent à penser que les techniques élaborées et expérimentées dans cette bibliothèque ouvrent d'intéressantes perspectives dans le domaine du calcul irrégulier parallèle (qu'il s'agisse de distribuer des structures de données irrégulières, et/ou d'appliquer des schémas d'accès irréguliers à des structures distribuées) et dans celui du traitement dynamique de la distribution et du parallélisme.

Nous souhaiterions en particulier étudier dans quelle mesure il est possible d'affaiblir les hypothèses de régularité qui sont à la base de la plupart des travaux menés actuellement dans le domaine du calcul massivement parallèle.

Les perspectives que nous entrevoyons à l'heure actuelle peuvent être classées en trois catégories principales, qui font l'objet des trois paragraphes suivants. Dans le paragraphe 7.2.1, nous envisageons la poursuite des expérimentations menées avec la bibliothèque Paladin à travers son utilisation interactive, son extension vers le domaine du calcul creux, et son utilisation comme support d'expérimentation pour développer des techniques d'optimisation globale des applications s'appuyant sur la redistribution des données. Dans le paragraphe 7.2.2, nous évoquons les possibilités de transfert des techniques dynamiques de distribution et de parallélisation élaborées dans Paladin vers le compilateur-paralléliseur Pandore (outil développé dans le cadre du projet Pampa). Dans le paragraphe 7.2.3, nous discutons des possibilités

1. *External Research and Development Program.*

de développement d'agrégats distribués irréguliers dans un environnement de type EPEE.

7.2.1 Perspectives d'expérimentation avec Paladin

7.2.1.1 Utilisation interactive

La gestion de la distribution des données et celle du parallélisme afférant sont, dans notre approche, totalement dynamiques. Cette caractéristique distingue fondamentalement les services offerts par Paladin de ce que peuvent offrir les compilateurs-paralléliseurs pour langages de type HPF [74, 8], avec lesquels la distribution et la parallélisation ne peuvent être résolus efficacement qu'à la compilation, et ce à condition que la distribution soit connue statiquement (dès qu'il y a redistribution dynamique et/ou appels de procédures, les compilateurs ne sont pas en mesure d'optimiser le code parallèle généré).

La bibliothèque Paladin offre donc des perspectives intéressantes dans le domaine du calcul parallèle interactif. On pourrait doter la bibliothèque d'un interpréteur permettant de créer et de manipuler de manière interactive des matrices et vecteurs distribués. On disposerait alors d'un outil offrant le même type de service que les outils *Matlab* et *Scilab*, mais capable d'effectuer des calculs d'algèbre linéaire sur une machine parallèle (qu'il s'agisse d'une machine telle que la Paragon ou d'un réseau de stations de travail) tout en présentant à l'utilisateur une interface purement séquentielle.

Avec cet outil, l'utilisateur serait en mesure de manipuler des vecteurs et matrices de plus grande taille qu'il ne pourrait le faire avec un outil séquentiel traditionnel, le calcul parallèle permettant de conserver des temps de réponse raisonnables. Toutefois, pour qu'un tel outil présente un réel intérêt, il faudrait que les entrées-sorties puissent également être réalisées en parallèle. À l'heure actuelle, les capacités des machines parallèles demeurent, dans ce domaine, très hétérogènes. Plusieurs projets sont s'ailleurs en cours (*e.g.* [46]), qui visent à définir des interfaces d'entrées-sorties parallèles de haut niveau pour les architectures parallèles.

Une approche possible pour permettre l'utilisation interactive de Paladin serait de l'intégrer à l'environnement logiciel *Scilab* développé par l'INRIA. Cet environnement, dédié au calcul scientifique, comprend notamment des modules pour le calcul d'algèbre linéaire. Il serait sans doute envisageable d'intégrer Paladin dans *Scilab* de telle sorte que l'interface de *Scilab* soit préservée, Paladin assurant les calculs en parallèle de manière transparente.

7.2.1.2 Utilisation comme outil de prototypage

La bibliothèque Paladin pouvant être aisément étendue, elle constitue un environnement de prototypage idéal. On pourrait ainsi l'utiliser pour expérimenter de nouveaux schémas de partitionnement ou de placement des données, pour tester de nouveaux formats de représentation interne des agrégats distribués, ou bien encore de nouvelles techniques de parallélisation, de nouveaux paradigmes de communication, de nouvelles règles de localisation des calculs (alternatives à la règle des écritures locales utilisée actuellement dans Paladin et dans Pandore), etc.

Étant interfacée avec la bibliothèque de communication et d'observation POM, la bibliothèque Paladin bénéficie de toutes les facilités offertes par la POM pour la génération automatique de traces d'exécution. Les différentes expérimentations évoquées plus haut pourraient donc être évaluées grâce aux outils d'analyse de traces développés dans l'équipe Pampa.

Il est notamment envisageable d'utiliser Paladin afin d'expérimenter des techniques de parallélisation et de distribution susceptibles d'être ensuite incorporées dans le compilateur-paralléliseur Pandore développé dans le cadre du projet Pampa. Un projet est d'ailleurs en cours pour expérimenter dans Paladin des schémas de parallélisation basés sur la migration des données[2], schémas qui pourront ensuite être intégrés à l'outil Pandore.

7.2.1.3 Extension pour le calcul creux

Dans son état actuel, la bibliothèque Paladin ne permet de créer et de manipuler que des vecteurs et matrices denses. Elle couvre donc approximativement le même domaine d'application que la bibliothèque ScaLAPACK et les compilateurs-paralléliseurs pour langages de type HPF. Il serait certainement très intéressant d'enrichir la hiérarchie des classes de Paladin afin d'aborder le domaine du calcul d'algèbre linéaire sur matrices et vecteurs creux, car ce domaine d'application est particulièrement difficile à traiter avec les compilateurs-paralléliseurs pour langages de type HPF, et n'est pas couvert par ScaLAPACK.

La mise en œuvre de vecteurs creux et de matrices creuses, qu'ils soient locaux ou distribués, ne pose pas de problème majeur dans un environnement de programmation par objets aussi riche et varié que celui du langage Eiffel. Les classes de la bibliothèque standard d'Eiffel décrivent un très grand nombre de structures de données irrégulières et/ou dynamiques pouvant servir de supports de mise en œuvre aux matrices et vecteurs creux. On pourrait par exemple s'appuyer sur la classe standard FIXED_LIST pour représenter en mémoire sous forme de listes fixes[3] les

2. Sujet de DEA de B. Certain, proposé et encadré par J.-L. Pazat.

3. Les listes fixes sont des listes représentées sous la forme d'un tableau mono-dimensionnel.

vecteurs locaux creux lorsque le nombre d'éléments non nuls et la répartition de ces éléments ne varient pas — ou très peu — au cours de la vie d'un vecteur. Les autres vecteurs locaux creux pourraient être représentés en mémoire sous forme de listes dynamiques [4], cette structure étant décrite dans la classe standard DYNAMIC_LIST. Les matrices locales creuses pourraient quant à elles être représentées sous la forme de tables de vecteurs locaux creux, ou bien encore de vecteurs locaux creux de vecteurs locaux creux.

Une fois mis en œuvre les agrégats vecteurs et matrices locaux creux, l'obtention des agrégats creux et distribués est quasi immédiate : de même qu'on a représenté les matrices distribuées par lignes sous la forme d'une table de vecteurs locaux, on peut tout aussi aisément représenter une matrice creuse distribuée par lignes sous la forme d'une table de vecteurs creux locaux.

La représentation interne des vecteurs et agrégats creux, qu'ils soient locaux ou distribués, ne pose donc pas de problème majeur. Il est en revanche plus difficile de réaliser des calculs efficacement avec de tels objets. Les algorithmes séquentiels encapsulés dans les classes VECTOR et MATRIX jouent toujours bien leur rôle d'algorithmes par défaut : étant totalement indépendant du format de représentation interne des vecteurs et matrices manipulés, on peut les utiliser pour réaliser des calculs portant sur des vecteurs et matrices creux. Par contre, ces algorithmes étant basés sur des itérations traditionnelles, ils ne permettent pas d'observer des performances satisfaisantes avec des objets creux.

Pour améliorer ces performances, il faut mettre en œuvre des *itérateurs* appropriés. Un itérateur permet, par exemple, d'énumérer dans le sens des indices croissants ou décroissants (selon le choix exprimé) tous les éléments non nuls d'un vecteur creux. Un produit scalaire de deux vecteurs (dont l'un au moins est creux) peut donc être réalisé de manière efficace en énumérant les éléments non nuls du vecteur creux et en ne procédant aux calculs élémentaires du produit scalaire que pour ces éléments. Dans la classe SPARSE_VECTOR caractérisant les vecteurs creux, on pourrait ainsi redéfinir l'opérateur *dot* (calculant le produit scalaire de deux vecteurs) comme illustré dans l'exemple 7.1.

En développant des itérateurs distribués (capables d'énumérer sur chaque nœud les éléments non nuls *locaux* d'un vecteur creux distribué), il devient possible de dériver à partir des algorithmes séquentiels creux des algorithmes parallèles performants. Grâce au mécanisme de l'encapsulation, ces algorithmes peuvent en outre être intégrés à la bibliothèque Paladin, et les détails de leur mise en œuvre masqués à l'utilisateur.

Pour que le coût des communications ne soit pas trop pénalisant, il faudrait

4. Les listes dynamiques sont des listes chaînées, dont le nombre d'éléments peut diminuer ou augmenter dynamiquement.

Exemple 7.1

```
class SPARSE_VECTOR inherit
   VECTOR
      redefine dot end;

feature                                                                    5
   dot (B: VECTOR): like item is
      local
         i: INTEGER;
         v: like item;
      do                                                                   10
         from start until off loop    -- Enumerate non-zero elements
            i:= cursor.index;    -- Extract 'index' field from current elt
            v:= cursor.value;    -- Extract 'value' field from current elt
            Result:= Result + v * B.item (i);  -- Perform computation
            forth;    -- Jump to next non-zero element                   15
         end; -- loop
      end; -- dot
   ...
end -- SPARSE_VECTOR
```

faire des objets creux des objets *transmissibles*. Il suffirait d'implanter en tenant compte du format de représentation interne choisi pour les matrices et vecteurs creux les quelques routines de communication (*send*, *recv_from*, etc.) déclarées dans la classe TRANSMISSIBLE, de telle manière que l'émission d'un objet creux implique une phase d'agrégation des données transmises, et que la désagrégation ait lieu de même lors de la réception de ces données au niveau de chaque nœud destinataire. Grâce au mécanisme de l'encapsulation, ces opérations d'agrégation et de désagrégation peuvent être totalement masqués à l'utilisateur d'objets creux.

Pour évaluer les possibilités d'extension de Paladin dans le domaine du calcul creux, nous avons d'ores et déjà effectué quelques expériences préliminaires. Nous avons développé quelques classes décrivant des vecteurs et matrices creux, et avons développé des algorithmes parallèles capables de manipuler ces objets de manière efficace. Les résultats de ces premières expériences sont très encourageants. La figure 7.1 montre les accélérations observées en effectuant une factorisation de Cholesky et un produit de matrices sur un réseau de stations de travail (stations Sun). Pour ces mesures, les matrices opérandes étaient des matrices distribuées creuses de taille 1000×1000 et de densité 10.0 % (proportion d'éléments non nuls dans les matrices).

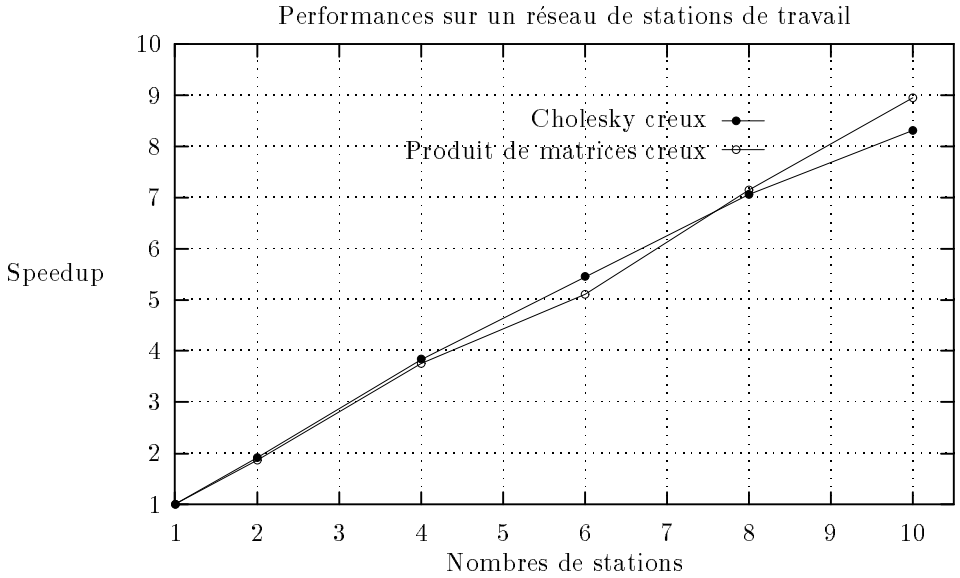


FIG. 7.1 - *Calculs parallèles creux (densité* 10.0 %*) réalisés sur un réseau de stations Sun*

7.2.1.4 Vers l'optimisation globale des applications

Le mécanisme de redistribution des matrices disponible dans la bibliothèque Paladin permet d'envisager l'optimisation globale d'un programme d'application. Il s'agit de permettre d'exploiter au mieux le polymorphisme des agrégats matrices dans un programme d'application, en les redistribuant chaque fois que le besoin s'en fait sentir.

Toutefois, le coût d'une redistribution étant loin d'être négligeable, il est indispensable de ne l'utiliser que de manière judicieuse. Le problème majeur est bien sûr celui de l'évaluation du coût relatif d'une opération et de celui d'une redistribution. Il faut être capable, connaissant les matrices manipulées dans chaque phase de calcul, d'exprimer le coût de cette opération dans une métrique appropriée (paramétrée notamment par les dimensions des objets manipulés et par leurs schémas de distribution respectifs). Il faut de même être capable d'exprimer dans la même métrique le coût de la redistribution d'une matrice, connaissant le schéma de distribution source et le schéma de distribution cible. Les difficultés principales sont ici :

- de trouver une métrique représentative du coût relatif de chaque opérateur et de chaque redistribution possible, les métriques communément utilisées pour exprimer le coût d'un algorithme (nombre d'opérations en virgule flottante, par exemple) étant ici totalement inappropriées ;

- d'évaluer effectivement ce coût pour chaque opérateur et chaque type de redistribution possible (les services d'observation offerts par la POM pourraient sans doute aider à cette évaluation) ;

- d'associer à chaque opérateur et à chaque routine de redistribution une fonction retournant le coût de l'opération considérée dans la métrique choisie.

En supposant résolu le problème de l'évaluation des coûts relatifs des distributions et des calculs, on peut alors envisager trois approches pour traiter le problème de la redistribution.

La redistribution « manuelle »

Avec cette approche, le mécanisme de redistribution est simplement mis à la disposition de l'utilisateur, qui demeure responsable du choix des schémas appropriés pour distribuer les agrégats matrices utilisés dans ses programmes d'application, et qui doit surtout déclencher explicitement la redistribution d'une matrice lorsque cela lui paraît judicieux. Cette approche présente l'avantage de pouvoir être immédiatement mise en œuvre, mais elle exige de la part de l'utilisateur une connaissance

approfondie de la bibliothèque Paladin. Il lui faut en effet savoir quels algorithmes parallèles ont été implantés dans la bibliothèque et avec quels schémas de distribution ces algorithmes peuvent atteindre de bonnes performances. On peut cependant envisager de mettre à la disposition de l'utilisateur les fonctions de calcul du coût évoquées plus haut en guise de mécanismes d'aide à la décision.

La redistribution « assistée »

Dans la redistribution « assistée », le programmeur d'application bénéficie de l'aide d'un outil interactif capable, au vu du code source du programme d'application, de déterminer quelles sont les distributions les plus appropriées pour les matrices utilisées et de suggérer au programmeur les redistributions adéquates. Cet outil interactif constitue une sorte de système expert, auquel on fournit toute l'information relative aux différentes distributions envisageables et aux exigences des opérateurs (avec pour chaque opérateur son ou ses schémas de distribution privilégiés).

La redistribution « automatique »

Les mécanismes de redistribution et de changement de type décrits au chapitre 5 font des agrégats matrices de Paladin des objets réellement polymorphes capables de changer de forme sans intervention explicite de l'utilisateur. En utilisant la routine *redistribute* évoquée dans le chapitre 5, on pourrait aisément faire en sorte que les opérateurs redistribuent eux-mêmes leurs opérandes, afin d'en adapter les schémas de distribution à leurs besoins propres. Si tous les opérateurs de Paladin étaient mis en œuvre de la sorte, l'utilisateur n'aurait plus à se soucier des schémas de distribution. Chaque matrice serait redistribuée dynamiquement au moment requis pendant l'exécution d'un programme d'application.

Cependant, redistribuer une matrice est une opération fort coûteuse, et il est donc très improbable que l'approche consistant à redistribuer les matrices à chaque instant au cours d'une exécution mène à de bonnes performances. Un opérateur pourrait par exemple redistribuer une matrice alors que l'opérateur invoqué aussitôt après en rétablirait la distribution initiale.

En fait, redistribuer les opérandes au début de l'exécution d'un opérateur constitue une technique d'optimisation dont la portée peut être qualifiée de *locale* : l'opérateur redistribue une matrice pour l'adapter à ses besoins propres afin d'obtenir de meilleures performances dans l'exécution du calcul dont il a la charge.

Pour que la redistribution automatique permette d'observer de bonnes performances *globales*, il faut inévitablement procéder à une analyse *statique globale* du programme d'application, afin d'examiner toute la séquence des opérations devant

être réalisées. Connaissant cette séquence d'opérations, on peut alors essayer d'en minimiser le coût en insérant au besoin des phases de redistribution entre les phases de calcul. Si le coût d'une opération et celui d'une redistribution peuvent être évalués et exprimés dans une métrique quelconque (qui reste cependant à définir), le problème de l'optimisation globale se ramène alors à un problème de recherche du chemin de poids minimal dans un graphe dont les arcs correspondent aux phases de calcul et de redistributions et sont pondérés par le coût relatif de ces opérations.

7.2.1.5 Extension vers HPF

Les mécanismes de gestion de la distribution incorporés dans l'environnement EPEE permettent d'envisager la distribution de tableaux multi-dimensionnels. Bien que les classes encapsulant ces mécanismes soient pour l'instant dédiées à la distribution de structures mono- et bi-dimensionnelles (elles ont été conçues pour gérer la distribution des vecteurs et matrices de Paladin), on a montré dans le paragraphe 3.2.5 qu'elles peuvent servir de briques logicielles de base pour construire grâce au mécanisme de l'héritage multiple de nouvelles classes capables de gérer la distribution de structures à K dimensions. Les schémas de distribution pouvant être gérés grâce à ces classes s'apparentent aux schémas de distribution autorisés par la syntaxe du langage HPF [74] : la distribution est réalisée sur la base d'un partitionnement en blocs homogènes. On a également montré dans le paragraphe 3.2.5 qu'il serait tout à fait envisageable de mettre en œuvre un mécanisme d'alignement entre les descripteurs de distribution, qui jouent le même rôle dans EPEE que les *templates* dans le langage HPF. La mise en œuvre de tableaux distribués « à la HPF » est donc envisageable en réutilisant les mécanismes qui ont déjà servi au développement de Paladin.

7.2.2 Transfert d'expertise vers Pandore

Dans le domaine des compilateurs-parallèliseurs pour langages de type HPF, on sait générer du code efficace lorsque la distribution des données est connue statiquement. En revanche, lorsque la distribution ne peut être connue statiquement (lorsque les données sont redistribuées et/ou lorsqu'il y a appel de procédure), les techniques mises en œuvre échouent.

Certaines des techniques dynamiques mises en œuvre dans Paladin pourraient alors être utilisées. On pourrait ainsi envisager d'intégrer dans le compilateur Pandore les mêmes mécanismes qui servent dans Paladin à assurer la sélection dynamique transparente des algorithmes parallèles. Dans Pandore comme dans Paladin, on se trouve en effet confronté au même problème : un programmeur peut écrire plusieurs algorithmes réalisant tous le même calcul, chacun de ces algorithmes ayant

toutefois des exigences particulières concernant, par exemple, les schémas de distribution des objets impliqués dans le calcul. Dans Paladin, nous avons mis en œuvre des mécanismes qui permettent d'assurer en fonction des caractéristiques dynamiques des agrégats distribués la sélection transparente de l'algorithme le mieux adapté pour réaliser le calcul requis par l'utilisateur. Des mécanismes semblables pourraient être mis en œuvre dans Pandore afin d'assurer la sélection dynamique des routines HPF en fonction des schémas de distribution des tableaux passés en paramètres.

7.2.3 Distribution et parallélisation des agrégats irréguliers

L'expertise acquise au cours du développement de la bibliothèque Paladin doit pouvoir s'appliquer à d'autres types d'agrégats que les vecteurs et matrices. Les techniques de conception des agrégats distribués élaborées lors du développement de Paladin ne sont aucunement limitées à la seule conception de bibliothèques manipulant des structures de données régulières, ni à la parallélisation d'algorithmes réguliers, comme en attestent d'ailleurs les résultats d'expériences récentes portant sur la parallélisation d'un serveur de routage SMDS pour réseau ATM [57], et sur la distribution de graphes d'accessibilité dans l'outil OPEN/CÆSAR [1].

Il serait intéressant d'étudier plus en détails les problèmes posés par la distribution et la manipulation en parallèle d'agrégats irréguliers, tels que des arbres, des listes, des graphes, etc. Une manière possible d'aborder ce problème serait de procéder à la parallélisation des « composants logiciels de Booch », plus connus sous le nom de *Booch Components*.

Les *Booch Components* forment une bibliothèque de classes décrivant aussi bien des structures de données (ensembles et multi-ensembles, listes, dictionnaires, graphes, listes, cartes, files, anneaux, piles, chaînes et arbres) que des « outils » (filtres, algorithmes de *pattern-matching*, de recherche, de tri) et des supports de mise en œuvre (tables de hashage, dictionnaires, conteneurs, etc.).

Conçus et développés à partir de 1987 par Grady Booch [24], et commercialisés par la société *Rational Software* sous l'appellation *Rational Booch Components*, ces composants logiciels ont immédiatement acquis une réputation de qualité et de fiabilité qui leur vaut de faire à présent office de bibliothèques de référence dans le domaine de la programmation par objets. Les *Booch Components* sont aujourd'hui utilisés comme supports de développement dans plus de 500 organismes (instituts, laboratoires et sociétés de conception de logiciel) à travers le monde. Initialement

développés en Ada, les *Booch Components* ont depuis lors été traduits en C++, et plus récemment en Eiffel[5].

La parallélisation des *Booch Components* pourrait être réalisée dans le cadre de l'environnement EPEE, en appliquant les techniques élaborées au cours du développement de Paladin. Outre que ce travail permettrait d'aborder la distribution et la manipulation en parallèle d'agrégats irréguliers et d'étudier comment les techniques développées avec Paladin s'appliquent à ce type d'agrégats, il pourrait éventuellement faire à terme l'objet d'un transfert industriel : le développement de *Booch Components* parallèles présentant une interface séquentielle intéresserait sans aucun doute la communauté des utilisateurs des langages à objets.

5. Un sous-ensemble des *Booch Components* a été développé en Eiffel par la société Tower Technology [111], et peut à présent être livré avec l'environnement de programmation TowerEiffel.

Bibliographie

[1] P. Abaziou. *Parallélisation d'OPEN/CAESAR dans l'environnement EPEE.* Rapport de stage de DEA, IRISA, 1994.

[2] H. Abelson, G. Jay Sussman, et J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Mac Graw Hill Book Company, 1985.

[3] G. Agha et C. Hewitt. Concurrent Programming Using Actors. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 37–53, The MIT Press, 1987.

[4] R.J. Allan. *PARLANCE v2.0 User Manual: Numerical Algorithms on a Virtual Shared Data Space.* Technical memorandum, Daresbury Laboratory, Warrington, U.K., 1992.

[5] R.J. Allan. Toward a Parallel Computing Environment for Fortran Applications on Parallel Computers. *Theoretica Chimica Acta*, 84 :257–269, 1993. Special Edition on Parallel Computing.

[6] P. America. Pool-T : A Parallel Object-Oriented Language. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220, The MIT Press, 1987.

[7] E. Anderson et al. LAPACK : A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of Supercomputing'90*, 1990.

[8] F. André, O. Chéron, M. Le Fur, Y. Mahéo, et J.-L. Pazat. Programmation des machines à mémoire distribuée par distribution des données : langages et compilateurs. *Techniques et Sciences Informatiques*, Numéro spécial, "Langages à parallélisme de données", 12(5) :563–596, octobre 1993.

[9] F. André, M. Le Fur, Y. Mahéo, et J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *HPCN Europe'95*, LNCS, Springer Verlag, Milan, Italy, May 1995.

[10] F. André, J.-L. Pazat, et H. Thomas. Pandore : a System to Manage Data Distribution. In *ACM International Conference on Supercomputing*, June 11-15 1990.

[11] Pampa Irisa (article collectif). Un support d'exécution pour machines parallèles. In *Actes des 6èmes rencontres francophones du parallélisme*, pages 207–212, RenPar'6 (Éd. Luc Bougé), juin 1994.

[12] C. Bareau. *Distribution automatique de programmes séquentiels : étude structurelle et expérimentale.* Mémoire de thèse, IFSIC / Université de Rennes 1, juin 1995.

[13] C. Bareau, B. Caillaud, C. Jard, et R. Thoraval. Correctness of automated distribution of sequential programs. In A. Bode, M. Reeve, et G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, pages 517–528, LNCS 694, Springer Verlag, June 1993.

[14] A. Beguelin, E. Seligman, et M. Starkey. *Dome : Distributed Object Migration Environment.* Technical Report CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, May 1994.

[15] S. Benkner. *Vienna Fortran 90 and its Compilation.* Mémoire de thèse, Université de Vienne, septembre 1994.

[16] J.K. Bennett. The Design and Implementation of DistributedSmalltalk. In *OOPSLA'87 Proceedings*, October 1987.

[17] M. Benveniste et V. Issarny. *ARCHE : un langage parallèle à objets fortement typés.* Rapport de recherche 1646, INRIA, mars 1992.

[18] M. Benveniste et V. Issarny. *Concurrent Programming Notations in the Object-Oriented Language ARCHE.* Research report 1822, INRIA, December 1992.

[19] R. Bielak. Identity Crisis. *Eiffel Outlook*, 4(3) :10–11, January 1995.

[20] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, et D.A. Moone. Common List Object System Specification. *SIGPLAN Notices (Special Issue)*, 23, September 1988.

[21] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, et F. Zdybel. CommonLoops : Merging Lisp and Object-Oriented Programming. In Norman Meyrovitz, editor, *Proceedings of the Conference on Object-Oriented Programming System, Languages and Applications (OOPSLA'86)*, pages 17–29, ACM, November 1986.

[22] F. Bodin, P. Beckman, D. Gannon, S. Narayana, et S. Yang. Distributed pC++ : Basic Ideas for an Object Parallel Language. In *Proceedings of the First Annual Object-Oriented Numerics Conference (OONSKI'93)*, pages 1–24, April 1993.

[23] F. Bodin, L. Kervella, et T. Priol. Fortran-S : a Fortran Interface for Shared Virtual Memory Architectures. In *Proceedings of Supercomputing'93 (ACM), Portland*, November 1993. Also available as a research report 702, IRISA.

[24] G. Booch. *Object Oriented Design with Applications.* Benjamin Cummings, 1991.

[25] B. Caillaud. *Contribution à la modélisation du SPMD : distribution asynchrone d'automates.* Mémoire de thèse, IFSIC/Université de Rennes I, juin 1994.

[26] D. Callahan et K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, 2 :151–169, 1988.

[27] L. Cannon. *A cellular computer to implement the Kalman filter algorithm.* Mémoire de thèse, Montana State University, Bozeman, MN, 1969.

[28] L. Cardelli et P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4) :211–221, 1985.

[29] D. Caromel. A General Model for Concurrent and Distributed Object-Oriented Programming. In *Proc. of the Workshop on Object-Based Concurrent Programming*, SIGPLAN Notices, February 1989.

[30] D. Caromel. Concurrency and Reusability : From Sequential to Parallel. *Journal of Object-Oriented Programming*, 3(3) :34–42, September 1990.

[31] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9) :90–102, September 1993.

[32] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, 1992.

[33] R. Chandra, A. Gupta, et J.L. Hennessy. COOL : a Language for Parallel Programming. In D. Gelernter et al., editor, *Languages and Compilers for Parallel Computing (chapter 8)*, MIT Press, 1990.

[34] C. M. Chase, A. L. Cheung, A. P. Reeves, et M. R. Smith. Paragon : A Parallel Programming Environment for Scientific Applications Using Communication Structures. In *1991 International Conference on Parallel Processing*, 1991.

[35] O. Chéron. *Pandore II : un compilateur dirigé par la distribution des données.* Mémoire de thèse, IFSIC/Université de Rennes I, juillet 1993.

[36] J. Choi, J. Dongarra, R. Pozo, et D.W. Walker. ScaLapack : a Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, Mc Lean, Virginia. In *Frontiers of Massively Parallel Computers*, October 1992.

[37] J. Choi, J. Dongarra, et D. W. Walker. PB-BLAS : a Set of Parallel Block Basic Linear Algebra Subprograms. In *Proceedings of SHPCC, Knoxville*, 1994.

[38] J.-F. Colin et J.-M. Geib. Eiffel Classes for Concurrent Programming. In J. Bezivin et al. (eds.), editor, *TOOLS 4*, pages 23–34, Prentice Hall, 1991.

[39] C.W.Kessler. Symbolic Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations. In *Proceedings of the Working Conference on Programming Environments for Massively Parallel Distributed Systems*, IFIP WG 10.3, April 1994.

[40] T.R. Davis. C++ Objects that Change their Types. *Journal of Object-Oriented Programming (JOOP)*, 5(4) :27–32, July 1992.

[41] J.W. Demmel, M.T. Heath, et H.A. van der Horst. *Parallel Numerical Linear Algebra.* Technical Report UCB//CSD-92-703, University of California, 1992.

[42] T. Dennehy. Class Libraries as an Alternative to Language Extensions for Distributed Programming. In *Proceedings of the USENIX Symposium on Experience with Distributed and Multiprocessor Systems*, pages 313–321, March 1992.

[43] M. Dion, J.-L. Philippe, et Y. Robert. *Parallelizing compilers : what can be achieved?* Research Report 94-11, École Normale Supérieure de Lyon, March 1994.

[44] J. Dongarra et al. An Object-Oriented Design for High-Performance Linear Algebra on Distributed Memory Architectures. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, 1993.

[45] T. Eggenschwiler et E. Gamma. ET++ Swaps Manager : Using Object Technology in the Financial Engineering Domain. In *Proceedings of OOPSLA'92, ACM SIGPLAN Notices, volume 27, number 10.*, pages 166–177, October 1992.

[46] P. Corbett et al. *MPI-IO : A Parallel File I/O Interface for MPI.* Research Report 19841 (87784), IBM T.J. Watson Research Center and NASA Ames Research Center, November 1994.

[47] R. Gabriel et al. CLOS : Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9), 1991.

[48] T. Fahringer, R. Blasko, et H. P. Zima. Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. In *6th ACM International Conference on Supercomputing*, pages 347–356, Washington, D.C., July 1992.

[49] P. Feautrier. Dataflow Analysis of Scalar and Array References. *International Journal of Parallel Programming*, 20(1) :23–53, February 1991.

[50] P. Feautrier. *Toward Automatic Distribution.* Technical Report 92.95, IBP/MASI, December 1992.

[51] J. Frankel. *C* language.* Thinking Machine Corporation, 1991. Reference manual.

[52] E. Gamma, R. Helm, J. Vlissides, et R.E. Johnson. Design Patterns : Abstraction and Reuse of Object-Oriented Design. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, pages 406–431, Springer-Verlag, Kaiserslautern, Germany, July 1993.

[53] A. Goldberg et D. Robson. *Smalltalk-80 : the Language and its Implementation.* Addison-Wesley, 1983.

[54] B. Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 165–176, ACM, Toronto (Canada), June 1991.

[55] G.H. Golub et C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1991.

[56] C. Gransart. *BOX : un modèle et un langage à objets pour la programmation parallèle et distribuée.* Mémoire de thèse, Université des sciences et technologies de Lille, 1995.

[57] F. Guerber, J.-M. Jézéquel, et F. André. *Conception et implantation d'un serveur SMDS sur architectures modulaires.* Publication interne 885, IRISA, novembre 1994.

[58] F. Guidec. Object-Oriented Parallel Software Components for Supercomputing. In *Proceedings of PARCO'95 (Parallel Computing), Gent, Belgium*, Universiteit Gent, 1995. À paraître.

[59] F. Guidec. Programmation par objets et parallélisme de données. In *Actes des 6èmes Rencontres Francophones du Parallélisme*, pages 187–191, RenPar'6 (Éd. Luc Bougé), juin 1994.

[60] F. Guidec. *Programmation par objets et parallélisme de données dans Paladin.* Publication interne 880, IRISA, octobre 1994. Également disponible en rapport de recherche INRIA, n^o 13307.

[61] F. Guidec et J.-M. Jézéquel. Design of a Parallel Object-Oriented Linear Algebra Library. In Proceedings of IFIP WG10.3. Karsten M. Decker et René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 359 – 364, Birkhäuser Verlag, Basel, July 1994. ISBN 3-7643-5090-3.

[62] F. Guidec et J.-M. Jézéquel. Embedding Data Parallelism in Sequential Object Oriented Languages. Position Paper for the Workshop on Distribution and Concurrency, TOOLS Europe'93, March 1993.

[63] F. Guidec et J.-M. Jézéquel. EPEE : An Environment for Parallel Execution of Eiffel. Invited Presentation at the International Eiffel User Conference (IEUC'93), March 1993.

[64] F. Guidec et J.-M. Jézéquel. *EPEE/P : an Eiffel Environment for the Paragon XP/S.* Technical Report 1.2, Irisa-Intel ERDP, June 1994.

[65] F. Guidec et J.-M. Jézéquel. *Experimenting numerical applications with EPEE/P.* Technical Report 1.1, Irisa-Intel ERDP, December 1994.

[66] F. Guidec et J.-M. Jézéquel. Numeric Parallel Programming with Sequential Object Oriented Languages. In *Proceedings of the First Annual Object-Oriented Numerics Conference (OONSKI'93)*, pages 55–69, April 1993.

[67] F. Guidec et Y. Mahéo. POM : une machine virtuelle parallèle incorporant des mécanismes d'observation. *Calculateurs Parallèles*, 7(2), 1995. Numéro spécial consacré aux environnements d'exécution de programmes parallèles.

[68] F. Guidec et Y. Mahéo. POM : a Parallel Observable Machine. In *Proceedings of PARCO'95 (Parallel Computing), Gent, Belgium*, Universiteit Gent, 1995. À paraître.

[69] F. Guidec et Y. Mahéo. POM : a Virtual Parallel Machine Featuring Observation Mechanisms. In *Proc. of the International Conference on High Performance Computing, New Delhi, India*, December, 27-30 1995.

[70] Y. Haddad. Performance dans les systèmes répartis : des outils pour les mesures. Thèse de Doctorat, Univ. Paris-Sud, Centre Orsay PARIS, septembre 1988.

[71] F. Hamelin, J.-M. Jézéquel, et T. Priol. A Multi-Paradigm Object Oriented Parallel Environment. In H. J. Siegel, editor, *Int. Parallel Processing Symposium IPPS'94 proceedings*, pages 182–186, IEEE Computer Society Press, April 1994.

[72] K.J. Hebel et R.E. Johnson. Arithmetic and Double Dispatching in Smalltalk-80. *Journal of Object-Oriented Programming (JOOP)*, 2(6) :40–44, March 1990.

[73] C.T. Ho, S.L. Johnsson, et A. Edelman. Matrix Multiplication on Hypercubes using Full Bandwitdh and Constant Storage. In *The Sixth Distributed Memory Computing Conf. Proc.*, pages 447–451, IEEE Computer Society Press, New York, 1991.

[74] HPF-Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[75] D.H.H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In Norman Meyrovitz, editor, *Proceedings of the Conference on Object-Oriented Programming System, Languages and Applications (OOPSLA'86)*, pages 347–349, ACM, November 1986.

[76] C. Jard, T. Jéron, G.-V. Jourdan, et J.-X. Rampon. A General Approach to Trace-Checking in Distributed Computing Systems. In 14^{th} *International Conference on Distributed Computing Systems, Poznan, Pologne*, pages 396–403, IEEE Computer Society Press, June 1994.

[77] C. Jard et J.-M. Jézéquel. ECHIDNA, an Estelle-Compiler to Prototype Protocols on Distributed Computers. *Concurrency Practice and Experience*, 4(5) :377–397, August 1992.

[78] C. Jard et G.-V. Jourdan. Incremental Transitive Dependency Tracking in Distributed Computations. *Parallel Processing Letters*, To be published, 1995. (Also available as a research report, IRISA 851).

[79] J.-M. Jézéquel. Building a Global Time on Parallel Machines. In *Proc. of the 3rd International Workshop on Distributed Algorithms*, pages 136–147, Lecture Notes in Computer Science, Springer Verlag, 1989.

[80] J.-M. Jézéquel. EPEE : an Eiffel Environment to Program Distributed Memory Parallel Computers. *Journal of Object Oriented Programming*, 6(2) :48–54, May 1993.

[81] J.-M. Jézéquel, F. Guidec, et F. Hamelin. Parallelizing Object-Oriented Software Through the Reuse of Parallel Components. *Object-Oriented System*, To be published, 1995.

[82] M. Karaorman et J. Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*, 36(9) :103–116, September 1993.

[83] A.H. Karp. Programming for Parallelism. *IEEE Computer*, 43–57, May 1987.

[84] M.F. Kilian. Object-Oriented Programming for Massively Parallel Machines. In *1991 International Conference on Parallel Processing*, 1991.

[85] E.K. Kolodner et W.E. Weihl. Atomic incremental garbage collection. In *Proc. Int. Workshop on Memory Management*, pages 365–387, Springer-Verlag, Saint-Malo (France), September 1992.

[86] Z. Lahjomri. *Conception et évaluation d'un mécanisme de mémoire virtuelle partagée sur une machine multiprocesseur à mémoire distribuée.* Mémoire de thèse, IFSIC / Université de Rennes 1, 1994.

[87] Z. Lahjomri et T. Priol. KOAN : a Shared Virtual Memory for the iPSC/2 hypercube. In *CONPAR92/VAPP V, Lyon*, September 1992.

[88] C. Lawson, R. Hanson, D. Kincaid, et F. Krogh. Basic Linear Algebra Subprograms for Fortran. *ACM Transactions on Math. Software*, 14 :308–325, 1989.

[89] M. Lemke et D. Quinlan. P++, a Parallel C++ Array Class Library for Architecture-Independent Development of Structured Grid Applications. *Sigplan Notices*, 28(1), 1993.

[90] J. Li et M. Chen. The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, 13 :213–221, 1991.

[91] D.J. Lickly et P.J. Hatcher. C++ and Massively Parallel Computers. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, pages 271–283, April, 1993.

[92] B. Liskov et J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[93] Y. Mahéo. *Environnements pour la compilation dirigée par les données : supports d'exécution et expérimentations.* Mémoire de thèse, IFSIC / Université de Rennes I, juillet 1995.

[94] Y. Mahéo et J.-L. Pazat. Distributed Array Management Scheme for Data-parallel Compilers. In *5th Workshop on Compilers for Parallel Computers*, malaga, Spain, June 1995.

[95] E. Maillet et C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 1995. À paraître.

[96] S. Matsuoka et A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha, P. Wegner, et A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, MIT Press, 1993.

[97] B. Meyer. *Eiffel : The Language.* Prentice-Hall, 1992.

[98] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[99] S. Murer, Feldman, et C.-C. Lim. *pSather : Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation.* Technical Report TR-93-028, ICSI, Berkeley, Ca., June 1993.

[100] M. O'Boyle et G.A. Hedayat. Data Alignment Transformations to Reduce Communications on Distributed Memory Architectures. *IEEE*, 366–371, 1992.

[101] S. Omohundro et C.-C. Lim. *The Sather Language and Libraries.* Technical Report TR-92-017, ICSI, Berkeley, Ca., 1991.

[102] C. Pancake et D. Bergmark. Do Parallel Languages respond to the Needs of Scientific Programmers? *IEEE COMPUTER*, 13–23, December 1990.

[103] B. Plateau. *APACHE : Algorithmique Parallèle et pArtage de CHargE.* Rapport de recherche APACHE-RR01 (disponible sur http://www-lmc.imag.fr/APACHE/apache.html), IMAG-LGI, novembre 1994.

[104] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hang, et G. Fox. *Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse.* Technical Report CS-TR-93-32, University of Maryland, College Park, MD, April 1993.

[105] J. Ramanujam et P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 3(4) :472–482, October 1991.

[106] J.-L. Roch et D. Trystram. Méthodologies pour la programmation efficace d'applications parallèles. *Calculateurs Parallèles*, 6(4) :133–138, 1994.

[107] J. Rumeur. *Communications dans les réseaux de processeurs. Études et recherches en informatique*, InterEditions, 1994. Ouvrage collectif coordonné par Denis Trystram.

[108] H.W. Schmidt. Data-Parallel Object-Oriented Programming. In *Proceedings of the Fifth Australian Supercomputer Conference, Melbourne*, pages 263–272, December 1992.

[109] E. Sibayama et A. Yonezawa. Distributed Computing in ABCL/1. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 91–128, The MIT Press, 1987.

[110] V. S. Sunderam. PVM : a Framework for Parallel Distributed Computing. *Concurrency : Practice and Experience*, 2(4) :315–339, December 1990.

[111] *Tower Eiffel Booch Components and Motif Components.* Tower Eiffel Corporation, 1.4.3. edition, December 1994.

[112] H. Thomas, H. Sips, et E. Paalvast. A Taxonomy of User-Annotated Programs for Distributed Memory Computers. In *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992.

[113] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* Mémoire de thèse, Rice University, January 1993.

[114] L.G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8), August 1990.

[115] R.C. Whaley. *Basic Linear Algebra Communication Subprograms : Analysis and Implementation Across Multiple Parallel Architectures.* LAPACK Working Note 73, technical report, University of Tennessee, 1994.

[116] P.R. Wilson et T.G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings OOPSLA '89*, pages 23–36, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[117] M. Wolfe. *Optimizing Supercompilers for Supercomputers. Research Monographs in Parallel and Distributed Computing*, Pitman, 1989. ISBN 0-273-08801-7.

[118] J. Wolff von Gudenberg. Design of a Parallel Linear Algebra Library for Verified Computation. *Reliable Computing*, 1995. À paraître.

[119] Y. Yokote et M. Tokoro. Concurrent Programmig in ConcurrentSmalltalk. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 129–158, The MIT Press, 1987.

[120] Y. Yokote et M. Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *OOPSLA'86 Proceedings*, 1986.

[121] A. Yonezawa, J.-P. Briot, et E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *OOPSLA'86 Proceedings*, September 1986.

[122] A. Yonezawa, E. Sibayama, T. Takada, et Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 56–89, The MIT Press, 1987.

[123] H. Zima et B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[124] H.P. Zima, H.-J. Bast, et Michael Gerndt. SUPERB : a Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, (6) :1–18, 1988.

Annexe A

Mécanisme des « vues » dans Paladin

A.1 Principe

En développant la bibliothèque Paladin, nous avons introduit la notion de « vue » sur un agrégat. Une « vue » est un objet qui, au lieu de stocker ses propres données, se contente en fait d'accéder en lecture ou en écriture à des données appartenant à un autre objet.

Ainsi, dans Paladin, une vue est une instance de l'une des classes ROW, COLUMN, DIAGONAL, SUBVECTOR et SUBMATRIX (figure A.1). Les trois premières classes caractérisent des vues qui se comportent « comme des vecteurs », la dernière classe caractérisant quant à elle des vues qui se comportent « comme des matrices ».

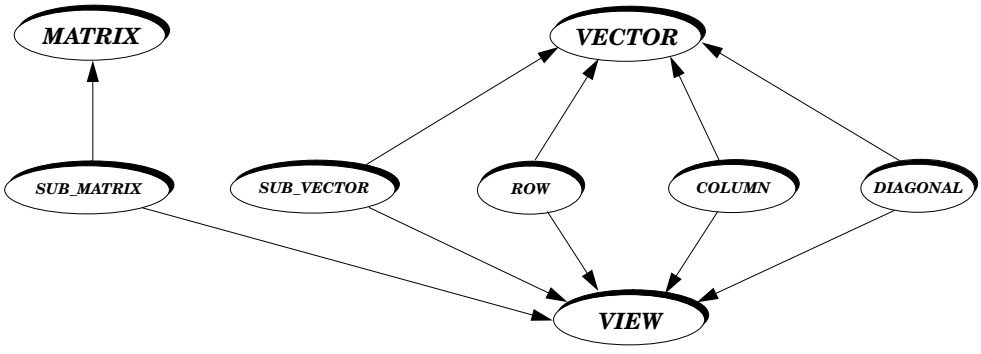


FIG. A.1 - *Les « vues » dans la hiérarchie de Paladin*

Une vue de type ROW, par exemple, est un objet qui se comporte « comme un vecteur » (la classe ROW hérite de la classe VECTOR), mais qui au lieu de stocker

directement ses propres données est capable d'aller lire ou écrire dans l'une des lignes d'un objet de type MATRIX. Pour permettre à un objet de type ROW d'accéder à une ligne quelconque d'une matrice donnée, il suffit que cet objet maintienne une référence vers la matrice considérée et le numéro de la ligne visée.

Les vues de type COLUMN et DIAGONAL sont bâties selon le même principe : elles maintiennent simplement une référence vers la matrice considérée et, respectivement, le numéro de la colonne ou de la diagonale visée.

Les vues de type SUB_VECTOR, qui permettent de manipuler un sous-vecteur comme s'il s'agissait d'un vecteur à part entière, doivent maintenir une référence vers le vecteur considéré et un couple (i_{min}, i_{max}) caractérisant la section contiguë visée dans ce vecteur.

Enfin, les vues de type SUB_MATRIX, qui permettent de manipuler une sous-matrice (section rectangulaire d'une matrice donnée) comme s'il s'agissait d'une matrice à part entière, maintiennent une référence vers la matrice considéré et un quadruplet $(i_{min}, j_{min}, i_{max}, j_{max})$ caractérisant la section visée dans cette matrice.

Les vues sont donc des objets particulièrement économiques du point de vue de l'occupation mémoire : elles n'occupent jamais plus de quelques octets, quelle que soit la taille du « vecteur » ou de la « matrice » qu'elles modélisent. Il faut en outre noter que les vues ne doivent pas nécessairement référencer un objet « concret » (c'est-à-dire un objet qui, lui, n'est pas une vue). Une vue peut très bien avoir pour objet de référence une autre vue. On peut ainsi par exemple manipuler comme un vecteur quelconque une vue, paramétrée pour désigner en fait un sous-vecteur d'une ligne d'une sous-matrice d'une matrice concrète...

A.2 Mise en œuvre

On a reproduit dans l'exemple A.1 la mise en œuvre de la classe ROW afin d'illustrer le principe de fonctionnement des vues.

Les classes COLUMN et DIAGONAL sont bien sûr mises en œuvre de manière très semblable. Les classes SUB_VECTOR et SUB_MATRIX sont un peu plus compliquées (on est amené à faire quelques calculs très simples portant sur les indices) mais cependant le principe de leur mise en œuvre demeure exactement le même que celui de la classe ROW.

- La classe ROW hérite de la classe VIEW et de la classe VECTOR. La classe VIEW est en fait une classe vide : elle permet simplement de distinguer dans la hiérarchie des classes de Paladin entre les classes qui définissent des vues et celles qui définissent d'autres types d'objets (voir figure A.1).

- Un objet de type ROW doit maintenir une référence vers un objet de type

Exemple A.1

```
class ROW inherit
    VIEW
    VECTOR
creation
    set_matrix                                                          5
feature -- Creation
    set_matrix (mat: like ref_matrix; i: INTEGER) is
        do
            ref_matrix := mat;
            ref_i := i;                                                 10
        end;
feature -- Attributes
    length: INTEGER is
        do
            Result := ref_matrix.ncolumn;                               15
        end;
feature -- Basic Accessors
    item (j: INTEGER): DOUBLE is
        do
            Result := ref_matrix.item (ref_i, j);                       20
        end;
    put (v: like item; j: INTEGER) is
        do
            ref_matrix.put (v, ref_i, j);
        end;                                                            25
feature {NONE} -- Private features
    ref_matrix: MATRIX;
    ref_i: INTEGER;
invariant
    valid_ref_matrix: (ref_matrix /= Void);                             30
    valid_index: (ref_i > 0) and (ref_i <= ref_matrix.nrow);
end -- class ROW
```

MATRIX ainsi que le numéro de la ligne visée dans cette matrice. Les attributs *ref_Matrix* et *ref_i* sont définis dans ce but (lignes 27 et 28) et sont en outre rendus « invisibles » pour l'utilisateur grâce à une clause d'exportation sélective (ligne 26).

- La matrice de référence ainsi que le numéro de la ligne visée doivent être spécifiés lors de la création d'une vue de type ROW. La routine de création (ligne 7) prend donc en paramètres les informations requises, et affecte les attributs *ref_Matrix* et *ref_i* en conséquence.

- L'invariant de la classe ROW garantit que chaque vue de type ROW est bien associée à une matrice existante et que la ligne visée existe bien dans cette matrice (lignes 30 et 31).

- En héritant de la classe VECTOR, la classe ROW hérite de tous les opérateurs et accesseurs qui y ont été définis. Il suffit donc, pour faire de la classe ROW une classe instantiable, de définir dans cette classe les deux accesseurs de base *put* et *item* ainsi que l'attribut *length* qui ont dû être maintenus différés au niveau de la classe abstraite VECTOR. La longueur d'un vecteur de type ROW est donnée par le nombre de colonnes dans la matrice de référence (ligne 15). Accéder en lecture ou en écriture à l'élément j d'un vecteur de type ROW équivaut à accéder à l'élément (ref_i, j) de la matrice de référence (lignes 20 et 24).

A.3 Exemple d'utilisation

Lors de la création d'un objet de type ROW, on passe en paramètres l'identité de la matrice de référence, ainsi que le numéro de la ligne visée dans cette matrice. En accédant à l'objet de type ROW, on accédera en fait aux informations stockées dans la matrice de référence.

Dans l'exemple A.2, on crée une une matrice M de taille 10×10 (le fait que M soit ici de type LOCAL_MATRIX n'a aucune espèce d'importance. Il pourrait tout aussi bien s'agir d'une matrice creuse, symétrique, distribuée, etc.). On crée ensuite une instance de la classe ROW, que l'on associe lors de sa création à la cinquième ligne de la matrice M. On peut ensuite manipuler la vue de la cinquième ligne de la matrice M, baptisée *my_row*, comme s'il s'agissait d'un vecteur à part entière. On peut donc par exemple invoquer sur cette vue quelques une des routines définies dans la classe VECTOR. Dans l'exemple A.2, on invoque ainsi l'opérateur *scal* afin de multiplier tous les éléments de *my_row* — c'est-à-dire en réalité tous les éléments de la cinquième ligne de M — par une valeur scalaire. On invoque ensuite

l'opérateur fonction *nrm2* afin qu'il retourne dans v la norme du vecteur constitué par la cinquième ligne de M.

Exemple A.2

```
local
    M : LOCAL_MATRIX ;
    my_row : ROW ;
    v : DOUBLE ;
do                                                    5
   !!M.make (10, 10) ;
   !!my_row.set_matrix (M, 5) ;
    ...
    my_row.scal (3.14) ;
    v := my_row.nrm2 ;                                10
end ;
```

A.4 Utilisation transparente des vues dans Paladin

En construisant les classes MATRIX et VECTOR, nous avons défini certains des accesseurs de ces classes afin d'exploiter directement le mécanisme des vues tout en rendant leur création transparente pour l'utilisateur..

Ainsi, dans la classe MATRIX l'accesseur vectoriel *row* est implanté de manière à créer et à retourner un objet de type ROW référençant la ligne spécifiée de la matrice courante.

Du point de vue de l'utilisateur, il importe peu de savoir que l'objet retourné par l'accesseur *row* est un objet de type ROW. Seul importe le fait que cet objet est d'un type conforme au type VECTOR (d'où la signature de cet accesseur en ligne 4 de l'exemple A.3), et peut donc être manipulé comme n'importe quel autre vecteur. (Les vues se distinguent cependant des autres vecteurs et matrices par une sémantique qui leur est propre. Cet aspect est discuté dans le paragraphe A.5.)

Les autres accesseurs de haut niveau de la classe MATRIX ont été mis en œuvre de manière semblable : les fonctions *column*, *diagonal* et *submatrix* retournent respectivement un objet de type COLUMN, DIAGONAL et SUBMATRIX. Dans la classe VECTOR, l'accesseur *subvector* est quant à lui défini de manière à retourner un objet de type SUBVECTOR.

Exemple A.3

```
deferred class MATRIX
    ...
feature -- Accessors
    row (i: INTEGER): VECTOR is
        -- Provide a view on i-th row
        require
            valid_i: (i > 0) and (i <= nrow)
        do
            !ROW!Result.set_matrix (Current, i);
        end;
    ...
end -- MATRIX
```

En définissant ainsi les accesseurs de haut niveau dans les classes MATRIX et VECTOR, on permet une utilisation transparente des vues dans un programme d'application. Ainsi, dans l'exemple A.4 on crée une matrice M de taille 10×10 et on calcule ensuite le produit scalaire[1] de la troisième ligne de M par la première diagonale de cette même matrice. En utilisant les accesseurs de haut niveau définis dans la classe MATRIX, on évite au programmeur d'application de devoir créer explicitement des objets de type ROW et DIAGONAL pour effectuer le calcul désiré.

Exemple A.4

```
        local
            M: LOCAL_MATRIX;
            v: DOUBLE;
        do
            !!M.make (10, 10);
            ...
            v := M.row (3).dot (diagonal (0));
        end;
```

1. L'opérateur fonction *dot* est défini dans la classe VECTOR. Il calcule et retourne le produit scalaire du vecteur courant et du vecteur passé en paramètre.

A.5 Sémantique des vues

Du point de vue d'un utilisateur de la classe MATRIX, l'objet retourné par l'accesseur *row* peut être perçu et manipulé comme un vecteur quelconque. Cependant l'utilisateur doit garder à l'esprit que s'il invoque sur cet objet des opérateurs ayant pour conséquence de modifier la valeur du vecteur considéré, c'est bien la matrice englobante qui va s'en trouver affectée. En d'autres termes, les accesseurs de haut niveau définis dans les classes MATRIX et VECTOR ne fournissent pas une *copie* de l'information désignée, mais un moyen d'accéder confortablement à cette information.

Si un utilisateur désire malgré tout disposer d'une copie de, par exemple, la septième colonne d'une matrice M quelconque, il peut aisément obtenir cette copie en créant explicitement un vecteur V de taille adéquate et en demandant explicitement la *copie* du contenu de la colonne visée dans V à l'aide de la routine *convert*, comme illustré dans l'exemple A.5.

Exemple A.5

```
local
    M : DROW_MATRIX ;
    V : LOCAL_VECTOR ;
do
   !!M.make (10, 10) ;                                    5
   !!V.make (10) ;
    ...
    V.convert (M.column (8)) ;
    ...
end ;                                                     10
```

Dans cet exemple, on cherche à obtenir une *copie* de la huitième colonne de la matrice M. On crée donc explicitement un vecteur V de taille adéquate, et l'on recopie dans ce vecteur le contenu de la huitième colonne de M en désignant cette colonne grâce à l'accesseur vectoriel *column* et en invoquant pour effectuer la recopie la routine *convert*[2].

2. La routine *convert* a été décrite dans le § 5.2.2. Elle permet de de recopier le contenu d'un vecteur dans un autre vecteur (ou celui d'une matrice dans une autre matrice). Elle est mise en œuvre de manière à être indépendante des formats de représentation de l'objet source et de l'objet cible.

Annexe B

Un mécanisme générique de réduction SPMD

B.1 Introduction

Les mécanismes de communication élémentaires offerts par la bibliothèque POM permettent de développer et d'encapsuler dans des classes des mécanismes génériques puissants pouvant aider à la distribution des données, aux mouvements de données, ou à la parallélisation des calculs. On décrit ici, à titre d'exemple, la mise en œuvre d'un mécanisme générique permettant de réaliser une opération de réduction SPMD.

B.2 Principe

On définit l'opération de réduction SPMD comme une opération impliquant

- un calcul *local* sur chacun des nœuds participant à l'exécution d'une application SPMD ;

- la « réduction » de l'ensemble des résultats locaux grâce à l'emploi d'une fonction binaire définie par une loi de composition interne (on supposera pour simplifier que cette loi de composition est commutative et associative).

Au terme de la réduction SPMD, le même résultat doit être disponible sur l'ensemble des nœuds participant au calcul.

B.3 Mise en œuvre

Caractérisation des fonctions de composition interne

Nous avons construit la classe BINOP afin de caractériser les opérations binaires définies par une loi de composition interne (exemple B.1).

Exemple B.1

```
deferred class BINOP [T]
feature
    op (a, b: T): T is deferred end;
end -- BINOP
```

La classe BINOP est générique et paramétrée par T (lors de l'instanciation, T peut prendre l'identité de n'importe quel type d'objet). Elle contient la déclaration d'une fonction binaire *op*, admettant en paramètres deux objets de type T et retournant un résultat du même type. Cette fonction est différée[1] : elle caractérise la famille de toutes les fonctions binaires définies par une loi de composition interne.

À partir de la classe BINOP, on peut construire à volonté des classes descendantes encapsulant chacune une définition possible de la fonction *op*. Par exemple, la classe SUM hérite de BINOP et définit la fonction *op* comme retournant la somme arithmétique de deux valeurs numériques (exemple B.2).

Exemple B.2

```
expanded class SUM [T -> NUMERIC] inherit
    BINOP [T]
feature
    op (a, b: T): T is do Result := a + b; end;
end -- SUM                                                    5
```

On pourra noter dans l'exemple B.2 que le paramètre générique T est ici contraint par le type NUMERIC, ce qui signifie que cette classe ne peut être instanciée qu'avec un paramètre formel de type INTEGER, REAL, etc. Sans cette exigence, l'expression de calcul de la somme arithmétique des termes a et b (calculée en ligne 4 dans l'exemple B.2) n'aurait aucun sens[2], et serait d'ailleurs refusée par le compilateur Eiffel.

1. Voir éventuellement le point de langage 2.3, page 49.

2. L'opérateur infixe d'addition « + » est défini pour tous les objets numériques, mais il ne l'est pas pour les tableaux, les listes, etc.

On pourrait construire un grand nombre de classes descendant de BINOP et encapsulant des fonctions arithmétiques (différence, produit, maximum, minimum, plus grand diviseur commun, etc.), des fonctions ensemblistes (union ou intersection d'ensembles), des fonctions de composition de listes, de graphes, d'arbres, etc.

Toutes les classes de ce type décrivent en fait des *agents*[3], pouvant être passés en paramètres à l'agent de réduction SPMD que nous décrivons à présent.

Abstraction algorithmique de la réduction SPMD

Nous avons construit une classe DIST_REDUCTOR encapsulant un algorithme de réduction SPMD (voir l'exemple B.3). Les instances de cette classe sont des *agents réducteurs*.

Exemple B.3

```
expanded class DIST_REDUCTOR [T]
feature
   reduce (v: T; action: BINOP[T]): T is
      local
         proc: INTEGER;                                          5
         tmp: T;
         POM: POM;
      do
         Result := v;
         -- Broadcast local value                               10
         POM.bcast (Result);
         -- Receive values from other nodes
         from proc := 0 until (proc = POM.nb_nodes) loop
            if (proc /= POM.node_id) then
               POM.recv_bcast_from (proc, tmp);                  15
               -- Invoke action on available values
               Result := action.op (Result, tmp);
            end; -- if
            proc := proc + 1
         end; -- loop                                            20
      end; -- reduce
   ...
end -- DIST_REDUCTOR
```

3. La notion d'objet agent a été introduite au paragraphe 3.2.4. Les agents sont des abstractions algorithmiques, c'est-à-dire des objets capables d'agir sur d'autres objets.

La classe DIST_REDUCTOR contient une seule routine, la fonction *reduce*, qui admet en paramètres une valeur v — le résultat du calcul effectué localement — et un objet agent *action* caractérisant l'opération binaire devant être utilisée pour effectuer la réduction.

Dans la routine *reduce* reproduite dans l'exemple B.3, nous nous sommes contentés d'implanter un algorithme très simple pour les besoins de l'illustration. Notre propos n'est pas ici d'obtenir un mécanisme de réduction SPMD générique optimal (à supposer d'ailleurs qu'un tel mécanisme puisse être construit indépendamment des caractéristiques de l'architecture cible), mais de montrer comment des mécanismes de ce type peuvent être développés et intégrés à la boîte à outils d'EPEE.

Dans l'exemple B.3, la réduction est réalisée en utilisant les services d'émission et réception en mode de diffusion offerts par la bibliothèque POM. Chaque nœud commence donc par diffuser la valeur locale v (ligne 11), puis il entreprend de recevoir toutes les valeurs émises par les autres nœuds (lignes 13 à 20). La réduction est réalisée en invoquant la fonction *op* de l'agent *action*, avec en paramètres le résultat intermédiaire de la réduction **Result**, et la nouvelle valeur *tmp* fraîchement reçue d'un nœud distant (ligne 17). La réduction se termine sur chaque nœud lorsque les contributions de tous les nœuds ont été prises en compte dans la réduction.

B.4 Exemple d'utilisation

On montre dans l'exemple B.4 comment un agent réducteur SPMD peut être utilisé pour réaliser une réduction sur des valeurs entières.

Exemple B.4

```
local
    sum_int: SUM [INTEGER];
    my_reductor: DIST_REDUCTOR [INTEGER];
    v, w: INTEGER;
do                                                              5
    v := { Actual computation }              -- <1>
    w := my_reductor.reduce (v, sum_int);    -- <2>
    ...
end;
```

Dans le petit programme SPMD de l'exemple B.4, on utilise un agent *sum_int* capable de calculer la somme de deux objets de type INTEGER, et un agent réducteur

my_reductor capable de procéder à des réductions SPMD sur des objets du même type.

Chaque nœud procède d'abord à un calcul local et place le résultat de ce calcul dans la variable locale v (phase $< 1 >$). La routine de réduction *reduce* est ensuite invoquée sur l'agent *my_reductor*, avec en paramètres la valeur de v et l'agent *sum_int* décrivant le type de réduction devant être effectuée. À l'issue de la phase $< 2 >$, la variable locale w a la même valeur sur tous les nœuds ayant participé à la réduction.

Résumé

Les méthodes et les environnements de programmation adaptés aux machines mono-processeur traditionnelles s'avèrent inutilisables avec les machines parallèles à mémoire distribuée, car ils ne permettent pas d'en maîtriser le parallélisme. À ce jour, l'utilisation de ces machines demeure donc très limitée, car les programmeurs sont en général assez réticents à l'idée de devoir y porter manuellement leurs applications.

De nombreuses recherches actuelles visent à simplifier le développement des applications parallèles pour ce type de machine. Le travail effectué au cours de cette thèse s'inscrit dans le cadre du développement et de l'expérimentation de l'environnement EPEE (Environnement Parallèle d'Exécution de Eiffel). EPEE constitue un cadre conceptuel pour la conception et la mise en œuvre de composants logiciels parallèles réutilisables à l'aide des mécanismes de la programmation par objets.

Nous avons caractérisé les objets pouvant être distribués et exploités en parallèle dans l'environnement EPEE, et proposé des schémas conceptuels permettant de développer de tels objets en insistant sur les points clés mis en avant dans les techniques modernes de génie logiciel, à savoir la maîtrise de la complexité (résolue par la modularisation, l'encapsulation, l'héritage), et la maintenabilité (corrective et évolutive).

Nous avons ensuite appliqué ces schémas conceptuels pour développer une bibliothèque parallèle de démonstration. Cette bibliothèque expérimentale, baptisée Paladin, est dédiée au calcul d'algèbre linéaire sur machines parallèles à mémoire distribuée. Elle est en outre extensible, d'un emploi aisé, performante et portable. Ces caractéristiques confirment la viabilité de l'approche consistant à utiliser les mécanismes de la programmation par objets pour construire des bibliothèques pour machines parallèles.